Configuration

Papers from the Workshop at ECAI 2000

21 – 22 August 2000

14th European Conference on Artificial Intelligence
Humboldt University
Berlin, Germany

Configuration

Papers from the Workshop at ECAI 2000

21 – 22 August 2000

14th European Conference on Artificial Intelligence
Humboldt University
Berlin, Germany

Organizing Committee

Markus Stumptner (Chair). Technische Universität Wien, Austria.
Michel Aldanondo. Ecole des Mines d'Albi Carmaux, France.
Gerhard Friedrich. University of Klagenfurt, Austria.
Esther Gelle. ABB Corporate Research Ltd, Switzerland.
Timo Soininen. Helsinki University of Technology, Finland.
Reijo Sulonen. Helsinki University of Technology, Finland.

Configuration Workshop at ECAI 2000

Program Committee

Michel Aldanondo, Ecole des Mines d'Albi Carmaux
Dean T. Allemang, Synquiry Technologies Ltd.
David Brown, Worcester Polytechnic Institute
Boi Faltings, Swiss Federal Institute of Technology
Gerhard Friedrich, University of Klagenfurt
Eugene Freuder, University of New Hampshire
Esther Gelle, ABB Corporate Research Ltd.
Albert Haag, SAP AG
Beat Liver, Credit Suisse Private Banking
Daniel Mailharro, ILOG S.A.
Hans Jørgen Skovgaard, BAAN
Timo Soininen, Helsinki University of Technology
Reijo Sulonen, Helsinki University of Technology
Markus Stumptner, Technische Universität Wien

Contents

Demo Descriptions

Acknowledgement: Final document preparation by Margit Rudelstorfer.

Preface

The ECAI 2000 Workshop on Configuration is the third in a series of international meetings on configuration that began with the 1996 AAAI Fall Symposium in Boston, USA, and continued with the AAAI'99 Configuration Workshop in Orlando, USA.

Configuration tasks can be defined as the design of an individual product using a set of pre-defined components or component types while taking into account a set of well-defined restrictions on how the components can be combined. This task can be supported by applying a wide range of AI techniques such as rule-based systems, constraint satisfaction and its extensions, description logics, logic programs, and different specialized problem solving methods.

Among the original application areas of the early expert systems, Configuration research has a long history, but has recently attracted a lot of research and industrial interest. The research interest is witnessed by the high turnout to events such as the AAAI'99 Workshop on Configuration. The industrial interest is indicated by the increasing number of vendors of software tools for configuration, i.e. configurators. The importance of configuration has expanded as more companies use configurators to efficiently customize their products for individual customers, and especially with the trend towards providing configuration support to the customers themselves (whether corporate or individual) via the Web.

However, efficient development and maintenance of configurators require sophisticated software development techniques. AI methods, more than ever, are central to developing powerful configuration tools and to extending the functionality of configurators. Configuration problems, like planning problems, can be seen on one hand as an interesting test-bed for novel AI techniques. On the other hand, configuration problems can serve as input for new research directions. The seventeen papers included in these pages, dealing with topics such as modeling, efficient algorithms, testify to the relevance and technical challenge presented by configuration problems.

The Organizers

General configurator requirements and modeling elements

Michel Aldanondo [1] - Guillaume Moynard [1,2] - Khaled Hadj Hamou [1]

Abstract. Our communication deals with general configurator requirements and generic modeling issues. Configurators are software that assist the configuration task. General configurators are edited by software producing companies and target the most frequent configuration situations and are supposed to be used by non-computer specialists. Our communication proposes a requirement analysis for this kind of configurator and some modeling elements based on the DCSP[1] formalism. Examples are given to illustrate our ideas.

1 INTRODUCTION

In order to improve their competitiveness, companies try to launch on the market products with customization capabilities. In order to do so, they include configuration software (configurator) in their information system. Two alternatives exist in terms of configurator: the design of a "specific configurator" matching exactly the needs or the acquisition of a "general configurator" provided by a software editor (see the surveys of Sabin in [2] and Gartner Group in [3]).

Our works deal with this second kind of solution ("general configurator") either for designing configurator (Caméléon TM configurator software see Véron in [4] or deploying this software in industry see Aldanondo [5]). Deploying a general configurator requires designing a model of the product with all its customization possibilities. As industrial configuration situations are very different, we are working on the setting of modeling tools, enabling us to deal with the most frequent configuration situations.

Our communication is divided as follows. First, after a short presentation of the configuration problem, we underline the main requirements of general configurator users. Then, we analyze the modeling requirements and try to derive a classification of industrial configuration situations. According to this classification, we propose and discuss some modeling elements. Examples coming from the Lapeyre Group (industrial carpentry) illustrate the communication.

2 CONFIGURATION PROBLEMS, CONFIGURATORS AND REQUIREMENTS

2.1 Main trends of the configuration problem

From all the previous works achieved concerning configuration, it seems that some common features defining configuring could be:

- hypothesis: a product is a set of components,
- given:
  - a generic model of a configurable product able to represent a family of products with all possible variants and options, in which a generic model is a set of components plus a set of various constraints,
  - a set of customer requirements, in which each requirement can be expressed by a constraint,
- configuring can be defined as "finding at least one component set that satisfies all the constraints".

These elements can be found in the definitions proposed by [6], [7], [8] or [9].

A configurator is a software that assists the person in charge of the configuration task. It is composed of a knowledge base that stores the generic model of the product and a set of assistance tools that help the user finding the solution or selecting components.

2.2 Two classes of configuration problems and relevant configurator solutions

From this definition, we can identify two main classes of configuration problems. These two classes are relevant to a kind of degree of complexity of the configuration problem and can be associated with the two kinds of configurators.

The first class is close to a product design activity and is mainly achieved by research and development team of companies. An order of magnitude of the cycle time to provide a solution could be between a week and a month. This is the case for complex product configuration as those described, for example, in [8], [10] or [11]. In most of these cases, a specific configurator is necessary. For that class of problems, the customer/supplier relation is more on the Business to Business side (B to B) and the number of configuration tasks that need to be achieved is rather small (no mass configuration, for example: from 1 to 10 configuration tasks per month).

The second class rather deals with the selling side and is conducted by sale team of companies. In that case, the cycle time order of magnitude for setting a solution is less than an hour and sometimes less than a minute when configuring on line on a web site for example. The configured products are generally more simple than in the previous case, some examples can be found in [1], [12] or [13] or in the web sites list [14]. This is the target of general configurators that assist the Business to Customer (B to C) customer/supplier relation and need to be able to support mass configuration or a great amount of configuration tasks (for example: 1 to 100 configuration tasks per day).

2.3 Requirements of the two classes of configurators

In any case, the basic common requirement, in terms of assistance, is to guarantee that the configured product is consistent with the generic model and the requirements, at the end of the configuration task.

2.3.1 Specific configurator requirements.

As they are specific by definition, these configurators need to be designed and maintained by computer science specialists. Therefore, the generic model of the product can take complex constraints into account and the configurator can provide various assistance tools. The resulting configurator can match the exact demand of the design team of the company.

(1) DRGI, Ecole des Mines d'Albi, Campus Jarlard, Route de Teillet 81013 Albi CT Cedex 09 - France
(2) Lapeyre-GME, 2-4 Rue André Karman, 93200 Aubervilliers, France

As the cycle time of the configuration task is not a strong constraint, interactive configuration is not a significant requirement and batch processing is possible.

As they are mainly used to assist design activities, the optimality of the proposed configured products is very often an important requirement that can be fulfilled thanks to batch processing.

2.3.2 General configurator requirements.

As they are designed and provided by software editors, these configurators just need to be tailored according to the needs of the company sale team. These configurators must propose an environment enabling a non-computer science specialist to describe the generic model of its product. For most configurators, the assistance tools provided cannot be modified in order to match the specific needs of the user.

Being almost used "in front" of the customer and almost in "real time", batch processing is not possible and interactive configuration is one of the strongest requirement.

Due to previous requirements, optimality of the solution can not be taken as a strong requirement.

Some second-rank assistance requirements more or less taken into account by the two kinds of configurators are among others: consistency recovering, explanation generation, generic model validation, configuration with progressive definition of customer requirements and ability to re-use generic model elements.

2.4 Conclusions

Our purpose is to propose and to discuss modeling elements enabling a non-computer specialist to describe generic models that must be set in a general configurator used in an interactive way. The next section will analyze generic modeling requirements.

3 GENERIC MODELING REQUIREMENTS

We first present what we call the "central problem" which is most of the time supported by all the general configurators. Then, each following section will describe an additional modeling requirement to the central problem. We finally conclude with some kind of cross analysis between these requirements and a classification of production situations. Our starting point is the definition elements of section 1.1.

3.1 The central problem

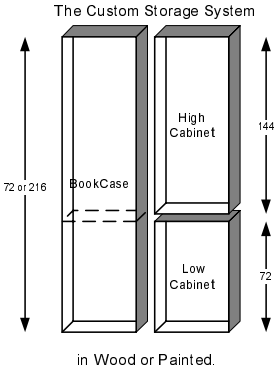
The first problem gathers the following elements:

- all the components are "standard" or completely defined, it is not possible to create a new component during the configuration task,
- the components are gathered in groups, each component must belong to only one group,
- the constraints represent the allowed combinations of components,
- the customer or configurator user requirement corresponds to the selection of one component in each group.
- a configured product is a component set, satisfying both constraints and requirements, where one and only one component must be selected in each group.

The purpose of the group notion is to gather components that support the same functionalities, therefore, one component must be selected in each group during configuration.

This problem is extremely rare. Very frequently, component group existence is optional or restricted for product feasibility reason. Therefore, two kinds of groups must be defined: the groups which always exist in any configured product and the ones that may exist according to customer requirement or product feasibility reason. This allows us to propose the hypothesis of what we call the central problem illustrated in figure 1:

- h1: all components are "standard" or completely defined, it is not possible to create a new component during the configuration task,
- h2: the components are gathered in groups, each component must belong to only one group,
- h3: a group is either always present in any configured product or its existence depends on:
  - (i) the existence of other groups,
  - (ii) the selection of other components,
  - (iii) configurator user requirements,
- h4: the constraints represent the allowed combinations of:
  - (i) component selection,
  - (ii) group existence,
- h5: the customer requirements correspond to:
  - (i) the selection of one component in each existing group (in some cases there must be only one remaining component that can be chosen),
  - (ii) the decision of a component group existence,
- h6: a configured product is a set of components satisfying both constraints and requirements, where one and only one component must be selected in each existing group.



Our central problem example is a simplified Custom Storage System issued from the Lapeyre Catalogue. 3 groups exist in our example : the BookCase (BC), the High Cabinet (HC) and the Low Cabinet (LC). The BookCase is available in two heights: 72 cm or 216 cm. All components exist in 2 finishes: Painted (P) or Wood (W). Therefore the groups of components are composed as follows:
- BookCase: {BC72P, BC72W, BC216P, BC216W}.
- High Cabinet: {HC144P, HC144W}.
- Low Cabinet: {LC72P, LC72W}.

In addition, some constraints are existing on those components:
- The assembly must be of the same color.
- The BookCase must be presented in all configured Storage Systems.
- The High Cabinet can exist if and only if a Low Cabinet exists and the BookCase is 216 cm high.

Figure 1. Central problem

3.2 Standard and "tailored component"

In the previous case, all the components are completely defined with entirely frozen characteristics. Very frequently, we have faced industrial cases where it was necessary to tailor components. For example when you arrange a piece of furniture in an old house, you need to have a specific size which rarely corresponds to the standard offer and may not match your "hole" in the wall. In order to capture this market, many companies propose customize possibilities that are not restricted to standard component assembling. The previous modeling hypothesis of the central problem become as follows and an example is shown in figure 2:

- h1.tc: all components are either "standard" or "tailored", it is not possible to create a new component during the configuration, a tailored component is characterized by numeric tailoring attributes with a continuous definition domain defining the range of possible values,

- h2: still valid,
- h3.tc: a group is either always present in any configured product or either its existence depends on:
  - (i), (ii), (iii) of list h3,
  - (tc-i) tailoring attribute value,
- h4.tc: the constraints represent the allowed combinations on:
  - (i), (ii) of list h4,
  - (tc-i) tailoring attribute value,

plus numerical constraints expressing a formula between tailoring attribute values,

- h5.tc: the customer requirements correspond to:
  - (i), (ii) of list h5,
  - (tc-i) the choice of a value for each tailoring attribute,
- h6.tc: a configured product is h6 plus values for tailoring attributes belonging to each selected tailored component.

To illustrate the problem of tailored components, we use the same example. However, the height of the BookCase and the High Cabinet can be tailored in a certain range:
- 72 ≤ Height(BC) ≤ 216,
- 50 ≤ Height(HC) ≤ 144.

The High Cabinet can exist if "Height(BC) ≥ 72 + 50 = 122". When the High Cabinet is existing, its top must be at the same height than the top of the BookCase. This needs the numerical constraint must be: "Height(BC)=Height(HC)+72".

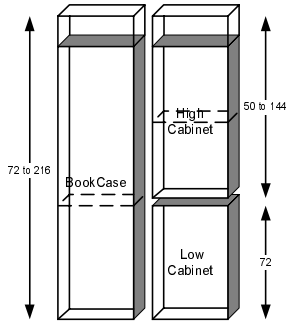


Figure 2. Tailored components

To illustrate this, we add 3 groups of components in 2 finishes to the example of figure 1: Painted (P) or Wood (W):
The Shelves: {SP, SW}.
The Drawers: {DP, DW}.
The Roll-Out Shelves: {ROSP, ROSW}.
In addition, we have the following constraints:
- If the BookCase is 72 cm high then you can place 0 or 1 Shelf in it, otherwise (when it is 216 cm high) you can put up to 3 Shelves.
- In the High Cabinet you have to choose 4 elements among Drawers and Roll-Out Shelves. At least one of each is necessary.

As before, the whole system must be of the same color.

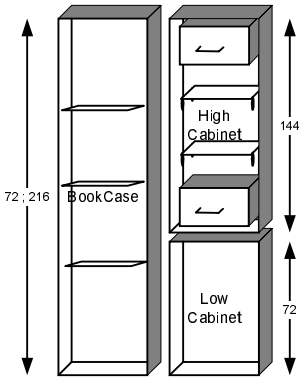


Figure 3. Component quantity

3.3 Component quantity

In the central problem, hypothesis h6 assumes that each selection action selects only one component. Very often it is necessary to model that the quantity of a selected component might be different from 1. For example, you may need to select a shelf quantity between 1 and 3 for a piece of furniture. The central problem hypothesis change as follows and figure 3 provides an example:

- h1: still valid,
- h1.qt: a standard component is, if necessary (default value is one), characterized by a quantity attribute which represents the possible quantities that can be selected,
- h2: still valid,
- h3.qt: a group is either always present in any configured product or its existence depends on:
  - (i), (ii), (iii) of list h3,
  - (qt-i) component quantity attribute value,
- h4.qt: the constraints represent the allowed combinations of:
  - (i), (ii) of list h4,
  - (qt-i) component quantity attribute value,
- h5.qt: the customer requirements correspond to:
  - (i), (ii) of list h5,
  - (qt-i) the choice of a quantity for each selected component,
- h6.qt: a configured product is h6 plus a quantity attribute value for each selected component.

3.4 Component loading

With the constraints defined in the central problem, it is delicate to model the fact that some component selections or group existence are restricted by some kind of resource (space, power, length…) provided by other components or groups. For example the component "BookCase of 72 cm high" allows the selection of 1 shelf, while the BookCase with 216 cm high allows to select up to 3 shelves. This need to define what we call a "numerical constraint" and modeling hypothesis h4 becomes:

- h4.cl: the constraints of h4 plus numerical constraints expressing a formula, function of (i) and (ii) of list h4, that must be true.

Most of the time, component loading constraints are used with component quantity attributes described in the previous section. For that case, component loading plus component quantity, hypotheses h4.cl becomes:

- h4.qt.cl: the constraints of h4 plus numerical constraints expressing a formula, function of (i), (ii) of list h4 and (qt-i) of list h4.qt, that must be true.

3.5 Component layout

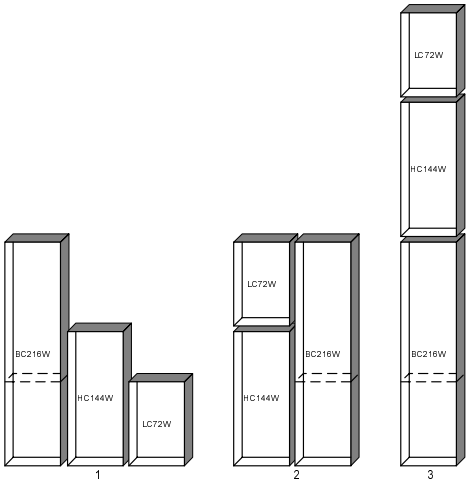
In the previous cases, the configuration result is always a set of components. In some cases, it is necessary to locate each selected component among others. We are closer to a design activity whose aims are either to locate a component in an absolute referential or to locate the position of one component relatively to another component. For more detail, see the elements reported by Brown in [15].

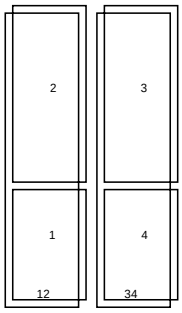
For the first case, absolute position (ap), hypotheses are as follows:

- h1: still valid,
- h1.ap: a component is characterized by numeric location attributes describing the possible locations of each component in an absolute referential,
- h2: still valid,
- h3.ap: a group is always present in any configured product or its existence depends on:
  - (i), (ii), (iii) of list h3,
  - (ap-i) component location attribute value,
- h4.ap: the constraints represent the allowed combinations of:
  - (i), (ii) of list h4,
  - (ap-i) component location attribute value,
- h5.ap: the customer requirements correspond to:
  - (i), (ii) of list h5,
  - (ap-i) the choice of a value for the location of each selected component,
- h6.ap: a configured product is h6 plus a value for the location of each selected component.

In the Central Problem, we describe the components (ie. BC, HC and LC) but we didn't locate them. Therefore, the layout could be:

Only the second one is valid, the BC must be under the LC and the BC near the LC.





The first solution, to overcome this problem, is to locate each component of the configuration into a zone: 1or 4 for LC, 2 or 3 for HC and 12 or 34 for BC.

The second solution, is to explain how to locate each component with respect to each other at a certain place (on, under, lined up, etc.).

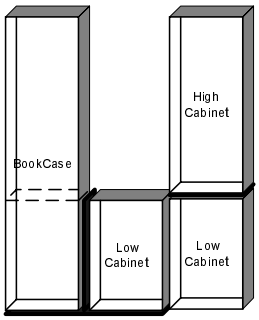


Figure 4. Component layout

For the second case, relative position (rp), hypotheses are as follows:

- h1: still valid,
- h1.rp: a component is characterized by attributes indicating the component places where one other component can be located or connected,
- h2 and h3: still valid,
- h4.rp: the constraints of h4 plus allowed couples of places relevant to couples of components that can be connected,
- h5.rp: the customer requirements correspond to:
  - (i), (ii) of list h5,
  - (ap-i) the choice of component places that must be used for connecting component couples,
- h6.rp: a configured product is h6 plus values for component places that must be used for connecting component couples.

Theses two cases are illustrated with the example of figure 4.

3.6 Conclusions

Each case has been defined by adding one requirement to the central problem, as shown in figure 5. Of course, when modeling Business to Customer industrial cases, all the requirement combinations can be met and the most complicated case gathers all the requirements. A similar conclusion is proposed in [16] where the central problem is called "feature-and-option configuration" and the remaining cases are gathered in "composition configuration" and "network configuration".

Case \ Section	2.1	2.2	2.3	2.4	2.5
Central problem	X	X	X	X	X
Tailored component		X			
Component quantity			X		
Component loading				X	
Component layout					X

Figure 5. Configuration requirements

Without configuration problem, some classification of industrial situations have been proposed by Apics [17] and Gartner [3] versus the customer order occurrence:

- pick to order (PTO):
  - components are available in inventory at the order occurrence,
  - there is no bill of materials defining the possible component combinations,
  - the order is a set of components chosen by the customer,
  - the supplier provides the components that will be assembled by the customer according to him.
- assemble to order (ATO):
  - components are available in inventory at the order occurrence,
  - there is a bill of materials for each possible product reference,
  - the order identifies a product reference chosen by the customer,
  - the bill of materials allows the supplier to launch assembly operations in order to provide the customer with a ready-to-use product.
- make to order (MTO):
  - raw materials are available in inventory at the order occurrence,
  - there is a bill of materials for each possible product and routing for each component,
  - the order identifies a product reference chosen by the customer,
  - the bill of materials and routings allow the supplier to launch components manufacturing and assembly operations in order to provide the customer with a ready-to-use product.

When configuration is added to these three situations, it is possible to talk about ~~a~~ "configure to order" (CTO) situations and the three situations correspond to the following classification:

- CTO-PTO: central problem - In that case configuration just prevents the customer from component gathering mistakes,
- CTO-ATO: pick to order plus component quantity and component loading - In that situation, there is no predefined bill of materials for each possible product, and the result of configuration should enable the supplier to derive the bill of materials definition that will be used for production management,
- CTO-MTO: assemble to order plus tailored component and product layout - Same as previous plus there is no predefined routing for tailored components and these routings must be derived from configuration results. Getting closer to a design activity, layout achievement belongs to this class of situation.

Most of the general configurator offer is centered on CTO-PTO and CTO-ATO, some of them tries to cover a little of CTO-MTO situations (mainly tailored products) figure 6.

The next section will try to provide modeling elements for these industrial situations and configuration requirements.

Case \ Situation	PTO	ATO	MTO
Central problem	X		
Tailored component			X
Component quantity		X	
Component loading		X	
Component layout			X

General configurator target

Figure 6. Requirements and production situations

4 MODELING ELEMENTS

As we are looking for modeling elements that will be used by non-computer specialists, the model must be easy to build and to understand. The expressiveness and understandability of DCSP [1] elements match this requirement. Our modeling discussion is therefore based on DCSP utilization for the various requirements presented in section 2.

4.1 Central problem modeling

The central problem can be modeled easily with the DCSP elements: variables, domains, compatibility constraints and activity constraints. The resulting model is what we call a "physical" model because each variable corresponds to a group of components as shown in figure 7 where the example of figure 1 is modeled.

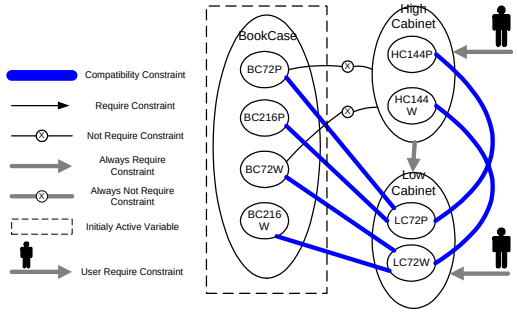


Figure 7. Central problem "Physical" model with DCSP

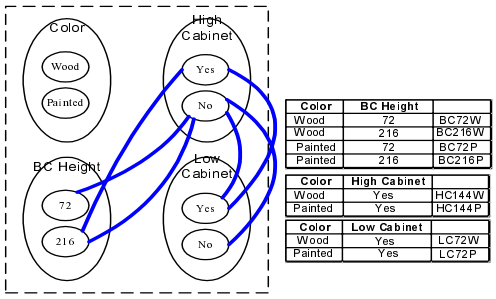


Figure 8. Central problem "Descriptive" model with DCSP

Very often, it is necessary to define "descriptive" variables that do not correspond to a component group. In that modeling approach, the values of descriptive variables permit to identify a component (for example the length and the width allow to identify a component shelf). In the work of Soininen [11] this is addressed as an explicit model (component oriented) versus an implicit model (description oriented). The advantages of the second approach are on the one hand that descriptive variables (fixed by the user) can be much more expressive for the user than a list of components (that do not interest the user most of the time) and on the other hand that configuration models can involve much less configuration variables and constraints. But the second approach needs to maintain identification tables allowing to derive the component list from the values of the descriptive variables. The example of figure 7 is presented in figure 8 with this second modeling approach.

4.2 Other requirements modeling discussion

This section reconsiders each requirement of section 2 and discusses how previous central problem modeling elements can match the modeling needs. The descriptive modeling approach is used to illustrate each case.

Tailored component requires tailoring attributes. Tailoring attributes correspond to descriptive attributes but their definition domains are continuous. Therefore, constraints dealing with tailoring attributes can be:

- when all the constraint variables correspond to continuous tailoring attributes, it is necessary to define numerical constraints with mathematical formula, in our example:
  (BC)Height = (HC)Height + 72,
- when the constraint gathers continuous tailoring attributes and other discrete variables, it is necessary to discretize on intervals each continuous tailoring attribute definition domain and to define compatibility and activity constraints on intervals, in our example:
  122>(BC)Height>72 compatible with
  HighCabinet = "Yes".

This permits to present in figure 9 the model of the example of figure 2.

Each component quantity attribute can be also taken into account as a descriptive variable. When component loading constraints are added to quantity attributes, two elements can be added to model this constraint:

- the resource / provide / consume elements as described in [10],
- a numerical formula between quantity attributes.

These two elements can model exactly the same loading constraint. The example of figure 3 is modeled in figure 10 with DCSP constraints (upper left) and a numerical constraint (lower right).

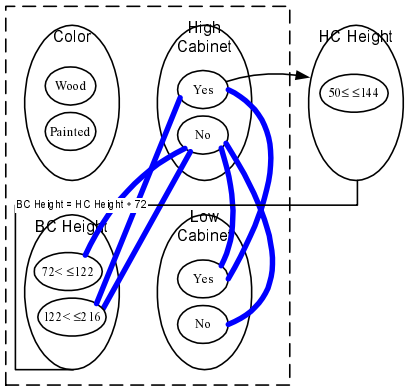


Figure 9. Tailored component model

Layout requirements using absolute position can be handled thanks to an association of each location attribute with a descriptive variable. If location attribute values are discrete and predefined, there is no specific modeling need. In the other case, when location attribute(s) is (are) defined on a continuous domain, it is necessary to use the modeling constraints proposed for tailored

attributes (numerical formulae and constraints with discretized intervals).

Layout requirements using relative location of component couple have been studied with the notion of "port" see [6]. Up to now, we have been able to add descriptive attributes to component in order to solve previous requirements and a configured product is a set of component plus values for descriptive attributes. Ports need to define more complex data structures and the resulting configured product cannot be a set of components plus a set of descriptive attributes values anymore. In order to handle such modeling, more complex to use tools are necessary as described in [8].

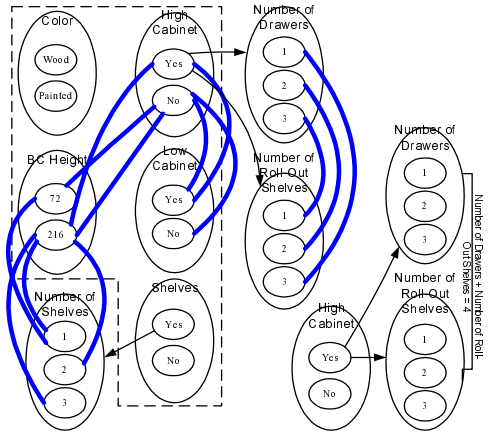


Figure 10. Component quantity and loading model

5 CONCLUSIONS

Starting from a central problem model composed of variables associated to component groups, compatibility and activity constraints; we now come with:

- variables associated with components group, descriptive attributes, tailoring attributes, quantity attributes, and location attributes,
- constraints, combining the previous variables, that:
  - can be compatibility constraints or activity constraints,
  - express allowed combination of values or numerical condition (numerical constraint with a mathematical formula or resource/provide/consume elements),
  - can be defined on discrete domains, continuous domains or continuous domain discretized in intervals.

Modeling complexity with the presented elements comes from the mix of all kinds of variables and the various constraints that can exist in a single problem. The big remaining resulting problem is model validation.

An important issue that has not been dealt with in this communication is that even for the central problem, it is very often required to have the configuration result (the component set) presented as a hierarchical bill of materials. In that case it is necessary to model the product as a generic hierarchical bill of materials, where the leaves are all the components that can be selected and the intermediate elements are generic sub- assemblies of the product. According to the configuration results the bill of materials links are validated or erased providing the hierarchical bill of materials of the configured product. Therefore, there is no modification of the central problem hypothesis and the configuration problem remains unchanged. Some elements about this hierarchical aspect can be found in [5] and [11].

Setting an easy to use modeling tool in order to "put on the paper" a generic model of a configuration situation is a big issue for configurator deployement in industry. Especially for what we call "general configurator", frendly user modeling is a necessity. With these modeling elements, presenting the important interest of being "visible" on a schema, we have been able to model a great variety of industrial configuration situations that did not require a specific configurator. For these cases, we have always succeeded to bypass component relative position layout modeling requirements, but we never faced the typical problem of hard electronics configuration problems needing connections and ports configuration.

6 REFERENCES

[1] S. Mittal, B. Falkenhainer, Dynamic Constraint Satisfaction Problems, Proceedings of the Ninth National Conference on Artificial Intelligence AAAI, 25-32, 1990.

[2] D. Sabin and R. Weigel, *Product Configuration Frameworks – A Survey. IEEE Intelligent Systems & their applications,* 13(4), pp. 42-49, July/August 1998.

[3] Gartner Group, *Sales Configurators: Configuring Sales Success, The report on Supply Chain Management, Advanced Manufacturing Research,* October 1997.

[4] M. Veron, M. Aldanondo and H. Fargier, *From CSP to Configuration Problems, AAAI Workshop on Configuration,* pp. 101-106, Orlando, Florida 1999.

[5] M. Aldanondo, S. Rougé and M. Veron, *Expert Configurator for Concurrent Engineering: Cameleon Software and Model, Special Issue: Production Systems Design and Control, Journal of Intelligent Manufacturing,* Vol. 11, n° 2, pp. 127-134, April 2000.

[6] S. Mittal and F. Frayman, *Towards a generic model of configuration tasks, International Joint Conference on Artificial Intelligence IJCAI,* Vol. 2, pp. 1395-1401, 1989.

[7] C. Kühn, *Requirements for Configuring Complex Software-Based Systems, AAAI Workshop on Configuration,* pp. 11-16, Orlando, Florida 1999.

[8] G. Friedrich and M. Stumptner, *Consistency-Based Configuration, AAAI Workshop on Configuration,* pp. 35-40, Orlando, Florida 1999.

[9] B. V. Faltings, R. Weigel, *Constraint-Based Knowledge Representation for Configuration Systems, Technical Report N° TR-94/59,* Laboratoire d'Intelligence Artificielle, Ecole Polytechnique Fédérale de Lausanne, Suisse, August 1994.

[10] D. Sabin, E.C. Freuder, *Optimization Methods for Constraint Resource Problems, AAAI Workshop on Configuration,* pp. 83-89, Orlando, Florida 1999.

[11] T. Soininen and al., *Modeling Configurable Product Families, 4° WDK Workshop on Product Structuring,* Delft University of Technology, The Netherlands, October 22-23, 1998.

[12] D. Sabin, E.C. Freuder, *Configuration as Composite Constraint Satisfaction, Proceedings of the Artificial Intelligent and Manufacturing Research Planning Workshop, AAAI Press,* pp. 153-161, 1996.

[13] T. Soininen, E. Gelle, *Dynamic Constraint Satisfaction in Configuration, AAAI Workshop on Configuration,* pp. 95-100, Orlando, Florida 1999.

[14] WWW Configurators
www.selectica.com
www.ikea.ca/content/furniture/designtools/design.asp

[15] D. C. Brown, *Defining Configuring, Artificial Intelligent for Engineering Design, Analysis and Manufacturing AIEDAM, Special Issue: Configuration Design,* 12(4), pp. 301-306, 1998.

[16] K. Orsvärn, T. Axling, *The Tacton View of Configuration Tasks and Engines, AAAI Workshop on Configuration,* pp. 127-130, Orlando, Florida 1999.

[17] APICS Dictionary, *The Educational Society for Resource Management, 8th Edition,* 1995.

Handling interactivity in a constraint-based approach of configuration

Jérôme Amilhastre [1] **and Hélène Fargier** [2]

Abstract. A configurable product can be represented by means of a Constraint Satisfaction Problem (CSP) the solutions of which are feasible products. Within an interactive configuration process, the final product cannot be generated by solving automatically the CSP; instead, the user specifies his requirements by the interactive choice of values for the different variables. The role of a decision support system is then to ensure at any time that the current set of user choices is consistent and to prune the domains of available values so as to keep them consistent with the choices. It also has to guide the relaxation by providing restorations, i.e. by identifying maximal consistent subsets of the user restrictions. This paper presents a framework to constraint-based configuration in which these functionalities are offered. It is based on an extension of the CSP framework in the spirit of ATMS. It relies on a pre-compilation of the initial CSP under the form of an automaton. This data structure is then used to maintain consistency and to compute restorations.

1 Introduction

Constraint programming techniques are widely used to model and solve decision problems. Most algorithms developed in this area focuses on the automatic solving of the problem. This does not help solving decision support problems that are interactive in nature. In this kind of application, the user himself chooses the values of the variables: the role of the system is not to solve a Constraint Satisfaction Problem, but to help the user in this task.

It is the case in the domain of product configuration: a *configurable product* is defined by a set of components, options, more generally by a set of attributes, the values of which have to be chosen by the user. These values must satisfy a set of configuration constraints that encode the feasibility of the product, the compatibility between components, their availability, etc. At a first glance, a configurable product can then be represented by means of a CSP[3]. The set of solution of this CSP represents the catalog, i.e. all the variants of the configurable product that the company proposes.

When configuring a product, the user specifies his requirements by the interactive choice of values for the different variables or more generally by the definition of *unary* constraints that restrict the possible values of the decision variables. After each new choice, the domains of the variables have to be pruned so as to guarantee that the values available can lead to a feasible product (i.e. a product satisfying all the constraints of the CSP). Finally, if the current set of choices becomes inconsistent with the constraints, or if the available values do not satisfy the user, he will backtrack on previous choices and relax some of them. Therefore the help provided by the decision support system should be to:

• **Maintain consistency**: The system has to ensure at any time that the current set of user choices is consistent with the CSP modeling the configurable product, or at least to detect inconsistency as soon as possible. It has also to compute the consequences of the actions of the user by deleting (resp. restoring) all the values that are incompatible (resp. compatible) with the current set of choices: the current domains should obey the property of "global consistency" or at least a property of local consistency, e.g. arc-consistency.

• **Guide relaxation on user requirements by providing restorations**: the system has to help the backtrack in answering the following questions of the user: "Which choices can I relax in order to recover consistency ?" or "Which choices can I relax in order to have this value available ?". The idea is thus to identify consistent subsets of the current set of user restrictions, possibly maximal consistent subsets or consistent subsets minimizing a cost function.

• **Provide explanations of the conflicts:** The system has to answer the questions "Which subset of restrictions do cause the inconsistency?" or "Why isn't this value available any more?". The idea is thus to identify (minimal) inconsistent subsets of restrictions.

Classical filtering algorithms, and above all, their dynamical versions (c.f. [1][2][6]) may address the first functionality, without providing any guarantee of global consistency. However, these algorithms cannot help the computation of restorations. This paper presents an approach of constraint-based configuration that addresses these functionalities. The next Section proposes an extension of the CSP framework that allows the modeling of the task of configuration. Our approach relies on the computation of an automaton that represents the initial CSP, as proposed by [13]. . The principles of this compilation are briefly recalled in Section 3. In Section 4, we present how this data structure is used to perform the task of consistency maintenance and to provide the user with restorations.

2 A constraint-based approach of configuration

2.1 Modeling the configurable product

A CSP Π is classically defined by a triplet $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ where $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ is a finite set of variables, each X_i taking its values in a finite domain D_{X_i} (where $\mathcal{D} = D_{X_1} \cup D_{X_2} \cup \ldots \cup D_{X_n}$), and a set of constraints $\mathcal{C}$. A constraint C in $\mathcal{C}$ is defined on $V(C) \subseteq \mathcal{X}$

[1] LIRMM, 161 rue Ada, 34392 Montpellier, France, amilhast@lirmm.fr
[2] IRIT, 118 route de Narbonne, 31062 Toulouse, France, fargier@irit.fr

[3] This is obviously an approximation: a product is often structured into subcomponents, the existence of which depends on the value given to some of the variables of the upper component. Several authors have proposed to extend the CSP framework in order to handle such structural characteristics [9, 7, 10, 11]. However, these works do not address the interactivy of the configuration task, and assume that the requirement of the user are completely given before the solving phase. The two ways of research (interactivity and structuration) are rather complementary than opposite.

and restricts the combinations of values that can be taken by the variables of $V(C)$. A relation $R(C)$ on $V(C)$ can be associated with each C: it is the set of tuples that satisfy the constraint. In the following, $\neg C$ denotes the constraint on $V(C)$ that is satisfied by any tuple that violates C and violated by any tuple that satisfies C.

An instanciation s is an element of $\mathcal{D}^n$; it is a solution of the CSP if it satisfies all the constraints, i.e. if, for any constraint C, the projection of s on $V(C)$ is an element of $R(C)$ (otherwise, s is said to falsify C). If such a solution exists, Π is said to be consistent, otherwise it is inconsistent.

In constraint-based configuration, the idea is to encode the configurable product by a CSP. Since each solution of the CSP corresponds to a variation of the product, we can reasonably suppose that it is consistent. Since it is possible to realize an off line pre-computing on the CSP (this pre-computing is not realized during the configuration phase, but takes places in the design of the configurator, once the generic product has been modeled by the expert), we can suppose that the initial CSP is *globally consistent*, i.e. that for any value in the domain of any variable, there is a solution that assigns this value to the variable. One of the tasks of the system will be to keep the problem globally consistent during the configuration phase, i.e. after each addition or deletion of a user choice.

2.2 Modeling the requirements of the user

The choices of the user deal with individual variables: an elementary choice is a unary constraint. This encompasses both the assignment of a precise value to a variable or the reduction of a domain to a subset of values that equally satisfy the user. Hence, we will represent the current set of user choices by a set $\mathcal{H}$ of unary constraints.

In this context, we can define the notion of "Assumption-based" CSP. This kind of CSP that relies on a distinguished set of constraints is an extension to non boolean domains of the so-called "Assumption based Truth Maintenance Systems", $\mathcal{H}$ playing the role of the set of assumptions:

Definition 1 *a* A-CSP Π *is a quadruplet* $< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ *where* $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ *is a CSP and* $\mathcal{H}$ *a set of unary constraints on variables of* $\mathcal{X}$ *.*

$< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ will represent the configurable product and $\mathcal{H}$ the current set of user restrictions. An instanciation s is a solution of Π if s is a solution of $\Pi' =< \mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} >$, which is the (classical) CSP associated with Π: Π' defines the feasible products that satisfies all the requirements of the user. Π is consistent (*resp.* inconsistent) iff Π' is consistent (*resp.* inconsistent). At any time, $\mathcal{S}(\Pi)$, the set of solutions of Π corresponds to the set of the products that obey the requirements of the user. For each X_i , P_{X_i} will be the projection of $\mathcal{S}(\Pi)$ on X_i : it is the set of all the values of X_i the choice of which allows the definition of a feasible product that satisfies $\mathcal{H}$.

Conflicts and consistent environments. In the following, we refer to subsets $E \subseteq \mathcal{H}$ of user choices as *environments.*

Definition 2 *Let* $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ *be a A-CSP. An environment is* consistent *(resp.* inconsistent*) with* Π *iff* $< \mathcal{X}, \mathcal{D}, \mathcal{C} \cup E >$ *is a consistent (resp. inconsistent) CSP. Any inconsistent environment is called a* conflict *for* Π *(or a conflict on* $\mathcal{H}$ *for* $\mathcal{C}$*).*

When $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ is consistent, a conflict can be understood as a cause of the inconsistency the inconsistency of the user requirements with the configuration constraints and any consistent environment defines a subset of $\mathcal{H}$ that can be relaxed so as to recover consistency: if E is a consistent environment, it is sufficient for the user to relax $\mathcal{H} - E$ in order to recover consistency.

For instance, if $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ is consistent and $< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ is inconsistent, $\mathcal{H}$ is a trivial conflict, $\emptyset$ a consistent environment and a dummy solution is obtained by relaxing all the constraint of $\mathcal{H}$.

This example also shows that all the environments are not equally interesting in practice. In real applications, the user choices do not necessarily have the same importance, but may be subject to preferences. For instance, a requirement dealing with the type of engine can be more important than a restriction concerning the color of the body of the car. In order to allow the handling of preferences, we propose to associate with each constraint $H \in \mathcal{H}$ a valuation $\phi(H) \in \mathbb{N}^+$: the higher the valuation, the more important the constraint. In the following, we suppose that every constraint has a positive importance (otherwise, it can be a priori discarded from $\mathcal{H}$). In this context, the best consistent environments ones are those that minimize the sum of the valuations of the relaxed constraints, or, equivalently, that maximize the sum of the valuation associated with the constraints they keep. In particular, when ϕ is the constant function 1, the best consistent environments are those that involve the larger number of constraints. Symmetrically, preferences can be used to discriminate the more or less interesting causes of the inconsistency, the best ones being the minimal ones: conflicts involving the less constraints will be preferred, or more generally the ones that minimize the sum of the valuations of the constraints involved. More formally:

Definition 3 *Let* $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ *be a A-CSP,* ϕ *a valuation of the assumptions, i.e. a positive application from* $\mathcal{H}$ *to* $\mathbb{N}$ *and for any* $E \in \mathcal{H}$*, let* $\phi(E) = \sum_{H \in E} \phi(H)$*.*

A V-optimal conflict *on* $\mathcal{H}$ *(or a* V*-nogood) is a conflict* E *of* Π *such that there is no other conflict* E' *of* Π *such that* $\phi(E') \prec \phi(E)$*.*

A V-maximal consistent environment *(or* V*-interpretation) of* Π *is an environment consistent with* Π *such that there exists no other environment* E' *consistent with* Π *and such that* $\phi(E') \succ \phi(E)$*.*

It could be interesting to use other structures of valuation as suggested in [12]. We restrict this paper to the structure $< \mathbb{N} \cup \{+\infty\}, >, + >$ with a ϕ giving valuations different from $+\infty$ and 0. This choice allows the representation of several optimality criteria for the environments, such as those based on cardinality or lexicographic ordering. It ensures a polynomial computation of the score of environments and thus a polynomial comparison between them.

Explanations and restorations Even if the current set of restrictions is not inconsistent with the CSP, it can forbid some values for other variables that the user would prefer to be available. The system must thus explain to the user a cause for these prohibitions:

Definition 4 *Let* $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ *be a A-CSP and* L *a unary constraint on a variable of* $\mathcal{X}$*.*

An explanation *of* L *on* Π *is an environment* E *such that* $< \mathcal{X}, \mathcal{D}, \mathcal{C} \cup E >$ *is a consistent CSP and* $< \mathcal{X}, \mathcal{D}, \mathcal{C} \cup E \cup \{\neg L\} >$ *is an inconsistent one.*

Determining why a set of values V is not available any more for a given variable X_i is indeed equivalent to determining which subsets of $\mathcal{H}$ are inconsistent with the constraint "$X_i \in V$", i.e. to compute the explanations of the constraint $L =$ "$X_i \in D_i / V$".

The user not only needs to understand why some values are forbidden, but also and most of all to know which previous choices can be relaxed in order to make these values available again. This leads to the definition of the notion of restoration:

Definition 5 *Let $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ be a A-CSP and L a unary constraint on a variable of $\mathcal{X}$. A* restoration *of L is an environment E such that $< \mathcal{X}, \mathcal{D}, \mathcal{C} \cup E \cup \{L\} >$ is a consistent CSP.*

Finally, we can again use the preferences expressed by ϕ to define the optimality of the explanations (the best ones being the minimal ones) and of the restorations (the best ones being those that keep the maximal number of important constraints). More formally:

Definition 6 *let $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ be a A-CSP, ϕ a valuation of the assumptions, i.e. a positive application from $\mathcal{H}$ to $\mathbb{N}$ and for any $E \in \mathcal{H}$, let $\phi(E) = \sum_{H \in E} \phi(H)$*

*A V-*optimal explanation *of L on $\mathcal{H}$ is an explanation E of L on $\mathcal{H}$ such that there exists no other explanation E' of L on $\mathcal{H}$ with $\phi(E') \prec \phi(E)$.*

*A V-*optimal restoration *of L is a restoration E of L such that there is no other restoration E' of L such that $\phi(E') \succ \phi(E)$.*

2.3 Summary of the tasks

In the context of a constraint-based and interactive approach of product configuration, we thus propose to model the configurable product by a CSP $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ and the current state of its solving by the user by means of a A-CSP $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$, where $\mathcal{H}$ stands for the current set of user restrictions. The preferences of the user are expressed by means of a valuation of these restrictions, i.e. a positive application ϕ from $\mathcal{H}$ to $\mathbb{N}^+$. In this context, the decision-support system is not in charge of an automatic resolution of the CSP but must rather perform the following tasks:

- Detection of the inconsistency: at any time, the system must tell whether there is a variant of the product that could satisfy all the requirements of the user, i.e. whether $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ is consistent or not
- Maintenance of the global consistency: at any time, the system must discard from the domains of available values those that cannot lead to a feasible product, i.e. compute P_{X_i} for any i.
- In case of inconsistency, computation of V-optimal nogoods and above all of guides for the restoration of the consistency, i.e. of V-optimal interpretations.
- When some interesting values become forbidden for a value, computation of V-optimal explanations of this fact and above all computation of V-optimal restoration of these values.

3 Compilation of a A-CSP

The A-CSP framework is direct transposition of ATMS to non-boolean domains; as a consequence, it define computational problems that are highly combinatorial The principle of our approach is to move the computational complexity of the handling of the on-line and repeated requests of the user to an off-line pre-computation; the idea is to compile the set of solutions of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ and to represent it by an automaton. The main interest of the compilation is that the interpretations and restorations can be generated from this form in linear time - linear in the size of the compiled form and, above all, *in the size of the result*. Moreover, this compilation is done only once, although several successive requests will be typically addressed in an interactive configuration situation, and that the system will used in several configuration situations.

This approach relies on a few hypotheses that are satisfied in practice by configuration problems: the persistence of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ and the possibility of performing off-line any computation on it, before the introduction of dynamic constraints; the structure of its set of solutions that allows its representation by an automaton - its size, exponential in the worst case, should be reasonable in practice; the restriction of the dynamic part of the problem, $\mathcal{H}$, to unary constraints.

3.1 Compilation of a CSP under the form of an automaton

Let us recall the keys points of the method introduced by Vempaty [13] to represent the set of solutions of a CSP as an automaton.

Automata We consider here the *state diagrams* of automata: a finite automaton (FA) $\mathcal{A}$ on an alphabet Σ is a oriented digraph the edges of which are labeled by elements of Σ (the *transitions* of the FA). In the following, $val(a)$ will denote the label of edge a. The finite set of its nodes (or *states* of the FA) is denoted Q. It has one *initial state* denoted I and at least one *final state*) F and is such that any edge and any state belongs to a least one path for the initial state to one final state. A word $m = a_1 a_2 a_3 ... a_n$ is recognized by the automaton if there exists one path from the initial state to a final state with the label m: $I \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} ... \xrightarrow{a_n} F$. The language recognized by $\mathcal{A}$ is the set $\mathcal{L}(\mathcal{A})$ of the words it recognizes. A FA is deterministic (DFA) iff all the edges coming from the same node have different labels.

Associating FA with CSP. Let $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ be a CSP. Given a permutation $O = [X_1, X_2, \ldots, X_n]$ of $\mathcal{X}$, any solution of Π defines a word of length n over the alphabet D. Hence, the set of solutions of Π defines a language over D. This language, called the solution language of Π w.r.t. O and denoted $\mathcal{S}_{\mathcal{O}}(\Pi)$, is a rational language. It is thus possible to represent $\mathcal{S}_{\mathcal{O}}(\Pi)$ by a FA. This automaton has only one final state (noted F) and is such that the length of any path from I to F is $n + 1$.

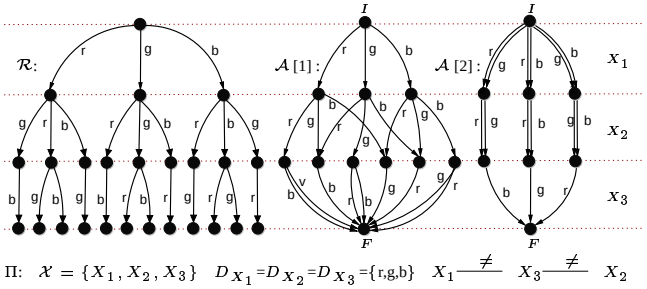


Figure 1. DFA $\mathcal{A}$ [1] and FA $\mathcal{A}$ [2] associated with Π for $O = [X_1, X_2, X_3]$ are concise representations of $\mathcal{R}$ the search tree of solutions

For any path $c = (I = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} q_3 \xrightarrow{a_3} ... \xrightarrow{a_n} F = q_{n+1})$ from I to F, for any state q_i in c, we write $var(q_i) = i$ (as soon as q_i is any state but F, $var(q_i)$ is the index of a variable in $\mathcal{X}$) and for any edge $a = (q_i, q_{i+1})$ of c, $var(a)$ is the variable $X_{var(q_i)}$ and $val(a)$ the label of a. Using these notations, $(val((q_1, q_2)), val((q_2, q_3)), \ldots, val((q_n, q_{n+1})))$ is thus an assignment of $(var((q_1, q_2)), var((q_2, q_3)), \ldots, var((q_n, q_{n+1})))$ and a solution of Π.

Finally, an edge a is said to *support* the value d for X_i iff $var(a) = X_i$ and $val(a) = d$. a is said to support a unary constraint L on X_i iff it supports at least one of the values of $R(L)$. A

path supports a value for a variable (resp. a unary constraint) iff it contains an edge supporting this value (resp. this unary constraint).

Computing the automaton associated with a CSP. Following [13], the automaton associated with a can be built by using classical operators on automata: either by any CSP algorithm that enumerates the set of solutions and adds them successively to the automaton, or by composition operators on (small) automata representing the configuration constraints. The complexity of the computation of the automaton and of its use depends on its size. This size is mainly influenced by (i) the order on the variables in the automaton and (ii) the algorithm used to built it. Vempaty [13] has proposed to build the DFA that minimizes the number of states. It is also possible to refer to other minimization algorithms, that build smaller automata (non-deterministic ones).

3.2 A-CSP with valuations and weighted automata.

Recall that each element of $\mathcal{H}$ is a unary constraint H_{X_i} on a variable X_i and is valuated by $\phi(H_{X_i})$; it forbids some assignments of X_i, and thus some of the edges: namely any edge a such that $var(a) = X_i$ and $val(a) \notin R(H_{X_i})$. The principle of our approach is to associate a weight to each edge: $\phi(a) = \phi(H_{X_i})$ if a corresponds to an assignment forbidden by a restriction H_{X_i}, $\phi(a) = 0$ otherwise. We can thus define:

- $p(c) = \sum_{a \in c_{q,q'}} \phi(a)$, the weight of a path c,
- $p(a)$ the weight of the best (i.e. with minimal weight) path from I to F through edge a,
- $p(q)$ the weight of the best path from I to F through state q

4 Answering the requests using the automaton

Now, since every constraint of $\mathcal{H}$ has a positive valuation, any path from I to F of zero weight (or "null path") defines a feasible product that satisfies all the user requirements and any non null path corresponds to a product that violates at least one of the restrictions of $\mathcal{H}$. Knowing the set of paths with a null weight is thus equivalent to knowing the set of solutions of $< \mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} >$.

Proposition 7 *Let $\Pi =< \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} >$ be a A-CSP, ϕ a positive valuation of its assumptions and $\mathcal{A}$ a weighted automaton representing Π. Let L be a unary constraint. It holds that:*

a) Any path c from I to F defines a consistent environment E such as $\phi(E) = \phi(\mathcal{H}) - p(c)$.
b) E is a consistent environment $\Rightarrow \exists$ a path c from I to F such that $p(c) \leq \phi(\mathcal{H}) - \phi(E)$.
c) Any path c from I to F that supports L defines a restoration E of L such a $\phi(E) = \phi(\mathcal{H}) - p(c)$.
d) E is a restoration of $L \Rightarrow \exists$ a path c from I to F that supports L such that $p(c) \leq \phi(\mathcal{H}) - \phi(E)$.

Sketch of proof:

- A path c from I to F defines a complete assignment of $\mathcal{X}$ that is a solution of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$. The weight of this path c is equal to the valuation the subset of constraints in $\mathcal{H}$ that are violated by this solution. Let us denote F this subset. $E = \mathcal{H} - F$ is thus a consistent environment. $\phi(E) = \phi(\mathcal{H} - F) = \phi(\mathcal{H}) - \phi(F) = \phi(\mathcal{H}) - p(c)$.
- Conversely let E be a consistent environment. There is a solution s of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ that satisfies all the constraints in E. Since all the solutions of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ are represented by the automaton, to s corresponds a path c. $p(c)$ is equal to the valuation of the subset of constraints in $\mathcal{H}$ that are violated by s. Let us denote F this subset: $p(c) = \phi(F)$. Since $F \subseteq \mathcal{H} \setminus E$, $p(c) \preceq \phi(\mathcal{H}) - \phi(E)$.
- A path c from I to F that supports L defines a complete assignment of $\mathcal{X}$ that is a solution s of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ and that satisfies L. $p(c)$ is equal to the valuation the subset of constraints in $\mathcal{H}$ that are violated by s. Let us denote F this subset. $E = \mathcal{H} \setminus F$ is thus a restoration of L . $\phi(E) = \phi(\mathcal{H} - F) = \phi(\mathcal{H}) - \phi(F) = \phi(\mathcal{H}) - p(c)$.
- Conversely let E be a restoration of L. There is a solution s of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ that satisfies L and all the constraints in E. Since all the solutions of $< \mathcal{X}, \mathcal{D}, \mathcal{C} >$ are represented by the automaton, to s corresponds a path c which supports L. $p(c)$ is equal to the valuation the subset of constraints in $\mathcal{H}$ that are violated by s. Let us denote F this subset: $p(c) = \phi(F)$. Since $F \subseteq \mathcal{H} \setminus E, p(c) \preceq \phi(\mathcal{H}) - \phi(E)$.

□

Proposition 7 allows the formulation in terms of optimal paths of most of the requests identified in Section 2.3. It indeed allows to prove that:

Proposition 8

a) Every minimal path from I to F represents a V-optimal interpretation of Π and conversely, to any V-optimal interpretation corresponds at least one minimal path.
b) $\mathcal{H}$ is a conflict for Π iff there is no null path from I to F.
c) $d \in P_{X_i}$ iff there is an edge a such that $var(a) = X_i, val(a) = d$ and a belongs to a null path from I to F.
d) Let L be a unary constraint on $X_i \in \mathcal{X}$, A_L the set of edges that support it and A^*_L the cheapest of them ($A^*_L = \{a \in A_L \ / \ p(a) = min_{a' \in A_L} p(a')\}$). Each edge in A^*_L defines a V-optimal restoration of L and to any V-optimal restoration of L corresponds at least one edge in A^*_L.

Hence, the requests we are interested in can be reduced to the determination of some minimal paths in the automaton. To allow an efficient computation of these minimal paths, let us maintain for any state q of the automaton:

- $p_g(q) = Min_{q' \in pred(q)}(p_g(q) + \phi((q, q'))$ is the minimal weight of the paths from I to q,
- $p_d(q) = Min_{q' \in pred(q)}(p_d(q') + \phi((q, q'))$ is the minimal weight of the paths from q to F,
- $p_g(I) = p_d(F) = 0$.

$p(q)$, the weight of the minimal path from I to F through q, and $p((q, q'))$ the weight of the minimal path from I to F through (q, q'), can be directly deduced from these quantities:

Proposition 9

- $p(q) = p_g(q) + p_d(q)$,
- $p((q, q')) = p_g(q) + \phi((q, q')) + p_d(q')$.

Finally, in order to ensure an efficient computation of the sets P_{X_i}, we maintain a counter $cnt(X_i, d)$ for any pair variable-value (X_i, d). $cnt(X_i, d)$ is the number of edges that (i) support the instanciation $X_i := d$ and (ii) belong to a null path from I to F.

4.1 Adding and deleting restrictions

Suppose that the user just adds to $\mathcal{H}$ a constraint H_i on X_i with the valuation $\phi(H_i)$. Some of the edges thus become forbidden: the a such that $var(a) = X_i$ and $val(a) \notin R(H_i)$. It is thus necessary to update the weight of these edges: since no other constraint of $\mathcal{H}$

restricts X_i, the weight of these edges must rise from 0 to $\phi(H_i)$. Conversely, if the user relaxes H_i and deletes it from $\mathcal{H}$, some of the forbidden edges become allowed: the edges a such that $var(a) = X_i$ and $val(a) \notin R(H_i)$. Since no other constraint of $\mathcal{H}$ restricts X_i, their weights come down to 0. In both cases, the modification of the weight of an edge has to be propagated back (to update the $p_d()$) and forward (to update the $p_g()$). This can be done in three steps:

1) Determination of the edges such that $var(a) = X_i$ and labeled by a value $val(a)$ that does not belong to $R(H_i)$; updating of their weights ϕ.

2) Backward propagation from the right states of these edges to the initial state: this is done with a breath-first strategy, so as to update the weights $p_d()$ of the traversed states knowing the weights of their successors.

3) Forward propagation from the left states of these edges to the final state: it is also done with a breath-first strategy, so as to update the left weights $p_g()$ of the traversed states knowing the left weights of their successors.

The $cnt()$ counters are maintained as follows: for any edge a encountered, $cnt(var(a), val(a))$ is incremented if $p(a)$ rises from 0 to a positive value and decremented if it comes down to 0.

This kind of algorithm is polynomial in the size of the automaton. Linear implementations can be proposed that rely on a judicious choice of the data structure that encodes the automaton. The practical efficiency of this kind of algorithm can obviously be enhanced, for instance by a propagation of the modifications only (in practice, the propagation has seldom to reach the extremities of the automaton).

4.2 Detection on the inconsistency

Maintaining the weighted automaton allows us to determine at any time whether the current set of assumptions (i.e. the current set of user choices) is consistent with the initial CSP. Indeed:

Proposition 10
- *$\mathcal{H}$ is a conflict iff $p_d(I) > 0$ (or $p_g(F) > 0$)*
- *$\phi(\mathcal{H}) - p(I)$ is the valuation of the V-optimal interpretations.*

4.3 Maintenance of the global consistency

According to proposition 8.c, P_{X_i} can be obtained from the automaton. It can actually be computed in linear time (linear in the size of D_{X_i}) using the counters $cnt()$:

Proposition 11 $P_{X_i} = \{d \ / \ cnt(X_i, d) > 0\}$

4.4 Computing the V-optimal interpretations

When the problem becomes inconsistent, the system must provide the user with V-optimal interpretations: they correspond to the minimal paths from I to F (*cf.* proposition 8.a). These optimal paths are easily obtained, going from I to F through "optimal" nodes, thanks to the following property:

Proposition 12 *c path from I to F with a minimal weight iff, for any edge (q, q') in c, $p_g(q) + p_d(q) = p_g(q') + p_d(q')$.*

The computational cost of a *unique* V-optimal interpretation is bounded by the number of variables and the maximal number of successors of a state. The search for *all* the V-optimal is more or less equivalent to the enumeration of all the minimal paths of the automaton, the difficulty being that a given interpretation can be represented by more than one path. Two different methods can be proposed for the enumeration of these interpretations. The first one is an adaptation of the DPI algorithm [4] that develops the tree of the V-optimal interpretations *without any backtrack due to a failure*. Indeed, the weight of the nodes can be used to perfectly determine whether an branch is optimal or not. The sketch of algorithms that follows is a simplified version that assumes that at most one constraint of $\mathcal{H}$ restricts a given variable X_i .

```
function optimal(q, q')
    return (p_g(q) + p_d(q) == p_g(q') + p_d(q'))

procedure Develop(i, list, E)
If i > n
    Memorize E (it is a V-optimal interpretation)
    return false                                                    (1)
If (there is in H a constraint H_i on X_i)
    Q_KeepH_i <-- {}
    Q_RelaxH_i <-- {}
    for all q in list do
        for all edge (q, q') such as optimal(q, q') do
            add q' to Q_RelaxH_i
            if φ((q, q')) = 0 then add q' to Q_KeepH_i
    If Q_RelaxH_i = {}
        return Develop(i + 1, Q_KeepH_i, E ∪ {H_i})
    Else, if Q_KeepH_i = {}
        return Develop(i + 1, Q_RelaxH_i, E)
    // thus Q_KeepH_i ≠ {} ∧ Q_RelaxH_i ≠ {};
    Else, if Develop(i + 1, Q_RelaxH_i, E)                          (2)
        return true
    Else,
        return Develop(i + 1, Q_KeepH_i, E ∪ {H_i})                 (3)
else // there is no constraint on X_i in H
    Q <-- {}
    for all q in list do
        for all edge (q, q') do
            if optimal(q, q') add q to Q
    Return Develop(i + 1, Q, E)
```

Several steps of the algorithm deserve some comments. A line (1), the search can be stopped as soon as some criterion is fulfilled, for instance when a given number of V-optimal interpretations have been reached: it is enough to return the value "true" instead of "false". At lines (2) and (3), the two branches of the search tree are developed. A more interactive solution should be to ask the user whether the branch "Relax H_i" should be developed or not. The important point is that a branch is entered (or proposed to the user) iff it is guaranteed that it leads to a V-optimal interpretation.

Another possible method is breadth searching the automaton, from I to F through the optimal paths. The principle is to label every state met with the sets of elements of $\mathcal{H}$ which must be relaxed to reach the state: we thus get on F the complementary (on $\mathcal{H}$) of each V-optimal interpretation. The most costly operation in this method is the updating of the sets that mark the states, since unioning sets is necessary at each updating. Anyway, for both methods, our representation by an automaton allows a computation of all the V-optimal interpretations that is polynomial in the size of the result.

4.5 Computing the V-optimal restorations

The automaton also allows the determination of the V-optimal restorations of a unary constraint L: they correspond to the cheapest path among those that go from I to F and that support L.

If the user is interested in a *unique* restoration of a set of values for a variable X_i , it is sufficient to find, among those supporting of these values, an edge (q, q') that minimizes $p((q, q'))$. Then the corresponding V-optimal restoration is obtained going backward from q to I and then forward from q' to F through an optimal path, thanks to the following property:

Proposition 13 *A path c from I to F that contains (q, q') is of minimal weight among the paths from I to F that contains this edge iff:*
- $\forall (e, e') \in c$ *s.t.* $var(e') \leq var(q), p_g(e') = p_g(e) + \phi(e, e')$
- $\forall (e, e') \in c$ *s.t.* $var(e) \geq var(q), p_d(e) = p_d(e') + \phi(e, e')$

The computational cost of a unique $V-$optimal restoration is thus bounded by the size of the automaton. Again, a judicious choice of the encoding of the automaton can lead to an implementation of this kind of algorithm that is bounded by the number of transitions that support L plus the product of the number of variables by the maximal number of successors of a state.

Now, in order to compute all the $V-$optimal restorations of L, we will re-use the principles presented in the previous section. The idea is to mark, among those supporting L, all the edges (q, q') that minimize $p((q, q'))$. The optimal paths can then be marked using property 13. It is then possible to reuse any of the methods of Section 4.4, rolled not on all the automaton but on the marked paths only. The worst computational cost for the generation of all the V-optimal restorations is again polynomially bounded in the size of the result.

5 Conclusion

Several authors [9, 7, 11] have proposed to extend the CSP framework so as to handle configuration problems - the extension dealing mainly with the difficulties that are inherent to the structuration of the problems. The handling of another characteristic of configuration problems, namely their interactivity, has prevailed on us to extend the framework in another direction and to define "Assumption-based CSPs". This led to an extension to non boolean domains of classical notions of propositional logic, e.g. nogoods, interpretations, explanations, etc.

The request associated with this generalization of ATMS obviously corresponds to problems that are highly combinatorial. From a practical point of view, our approach is to transfer this cost on an off-line pre-computation: the system works on a compilation of the CSP under the form of an automaton (the size of which can itself be exponential in the worst cases) but uses algorithms that are really efficient on this structure (polynomial in the size of the result). The way we use the automaton is close to the use of OBDD-like structures in the handling of prime implicant/implicate and interpretations of a boolean formula (see [8], for seminal works, and [5][3] for some approaches very similar to ours).

Our approach takes advantage of the fact that, in configuration problems, only a small set of constraints is subject to dynamicity, and that these constraints are unary. It relies on the fact that the components of configurable products can be described concisely by automata. The structuration of complex products into sub-components should now help in reducing the space needed to handle the configurable product. The next step of our research is to thus combine the representation by automata with composite CSP.

The principles proposed in the present paper have been implemented and tested on a preliminary benchmark coming from a real application in configuration: the size of the automaton is very small (about 4675 states + edges for a problem allowing 936 000 solutions), its updating and the handling of the requests is immediate (less than 2ms). We are presently encoding a bigger real example (about 200 variables). However, we will also have to face the definition of an experimental protocol for the evaluation of our work, that, even less close to real problems, could enlighten statistically the feasibility or the limits of the approach.

Finally we did not present algorithms dedicated to the computation of explanations and nogoods since this kind of information is generally less attractive than the notions of restoration / interpretation in the context of a configuration task [4], unless the number of explanations is small and the number of restorations great: in this case, it would be interesting for the user to build the restoration himself by selecting one constraint in each explanation. Anyway, the computation of nogoods and explanations can attractive for other potential applications of A-CSP. If we accept to relax the requirements of minimality and of completeness, an efficient approach to the generation of such information could take advantage of the justifications maintained by dynamic filtering algorithms like DN-AC4,6.

REFERENCES

[1] C. Bessière, 'Arc-consistency for dynamic constraint satisfaction problems', in *Proc. of the AAAI-91 conference*, pp. 221–227, (1991).

[2] C. Bessière, 'Arc-consistency for non-binary dynamic csps', in *Proc. of the ECAI 92, 10^{th} European Confernce on Artificial Intelligence*, pp. 23–27, (1992).

[3] Fabrice Bouquet and Philippe Jégou, 'Solving over-constrained csp using weighted obdds', *Over-Constrained Systems, LNCS 1106*, 293–308, (1996).

[4] T. Castell, 'Computation of prime implicates and prime implicants by a variant of the davis and putnam procedure', in *Proceedings ICTAI'96*, pp. 428–429, (1996).

[5] Claudette Cayrol, Marie-Christine Lagasquie-Schiex, and Thomas Schiex, 'Nonmonotonic reasoning: from complexity to algorithms', *Annals of Mathematics and Artificial Intelligence*, **22**(3-4), 207–236, (may 1998).

[6] R. Debruyne, 'Arc-consistency in dynamic csps is no more prohibitive', in *Proceedings of the eighth International Conference on Tools With Artificial Intelligence*, pp. 299–306, (1996).

[7] S. Mittal and F. Frayman, 'Dynamic constraint satisfaction problems', in *Proceedings AAAI'90*, (1990).

[8] J.C. Madre O. Coudert, 'A logically complete reasoning maintenance system based on a logical constraint solver', in *IJCAI'91*, pp. 294–299, (1991).

[9] Daniel Sabin and Eugene C. Freuder, 'Configuration as composite constraint satisfaction', in *Artificial Intelligence and Manufacturing Research Planning Workshop, AAAI Technical Report FS-96-03*, (1996).

[10] Mihaela Sabin and Eugene C. Freuder, 'Detecting and resolving inconsistency in conditional constraint satisfaction problems', in *Proceedings of the AAAI'99 Workshop on configuration*, (1999).

[11] Timo Soininen and Ester Gelle, 'Dynamic constraint satisfaction in configuration', in *Proceedings of the AAAI'99 Workshop on configuration*, (1999).

[12] Gérard Verfaillie Thomas Schiex, Hélène Fargier, 'Valuated constraint satisfaction problems : hard and easy problems', in *IJCAI95*, pp. 631–637, (1995).

[13] N. R. Vempaty, 'Solving constraint satisfaction problems using finite state automata', in *American Association for Artificial Intelligence (AAAI'92)*, pp. 453–458, San Jose, (1992).

[4] Their main interest is generally ... the deduction of restorations/interpretations by an operation of Hitting set, whereas the direct calculus is generally cheaper

Dealing with Uncertainty in Design and Configuration Problems

Eric Bensana and **Taufiq Mulyanto** and **Gérard Verfaillie** [1]

Abstract. Design is an activity that translate a set of requirements in a feasible product by considering constraints coming from physics, technology avaibility and designer preferences. Configuration is a special case of design in which the product component properties are already predefined. In this paper, the design of a new concepts (or the validation of existing ones) is considered as a configuration problem. In the first part of this paper, we present a generic approach for conceptual design, the earliest phase of design, combining the constraint satisfaction formalism and the object-oriented framework. In the second part, we introduce an approach to deal with uncertainties mostly found during the conceptual design phase. The proposed approach is very easy to implement and guarantees the upper bound of the solution existence probability. The designer can now make his or her decisions not only based on the product performance, but also on the risk associated to that performance.

1 Introduction

Design is an activity that translate a set of requirements in a feasible product which can best satisfy the realization constraints coming from physical phenomenas, the technology availability and designer preferences.

Configuration is a specific task of design. We can see a configuration task as a design of a product given a set of components (possibly sizable) described by a fixed set of properties including informations on the interactions within components. In this paper we restrict the design context in a configuration task.

A *concept* or a product is a solution of a configuration problem as a result of a *product instantiation*. In this paper, we are interested at a system to aid designer in the configuration task using a generic approach.

Our prior internal studies related to the design and sizing of sensing systems [3, 2] showed that the **CSP** framework [12] offers:

- a flexible approach by a natural problem representation using constraints;
- a context free application including an absence of domain type restrictions.

CSP framework is defined by a set of variables, a set of domains each associated to one variable and a set of constraints allowing values combinations within a sub set of variables. Domains can be of all types: symbolic or numeric, discrete or continue. Constraints can be *extensively* defined (list of allowed/prohibited value combinations) or *intensively* defined (relation such as equation). Further, a representation framework such as an object-oriented framework would enhance a design problem representation. We based our proposed generic approach on these two frameworks.

The generic nature of the proposed approach injures consequently its computing power, which limits its use for the *conceptual* design. More advanced phase, such as detailed design, specially in the aircraft design domain, is the realm of parametric optimization [10, 15] even if non classical optimization techniques such as genetic algorithms can be used [1].

Based on the CSP point of view, a number of frameworks have been proposed to deal with configuration problem.

The **Dynamic CSP** framework [14] extends the classical CSP framework to deal with a situation in which the solutions do not need to have the same variables and neither to satisfy the same constraints. The idea is to associate to each variable and each constraint two possible states: active and non-active. States control is done by a set of activation constraints.

The **Composite CSP** framework [19] answers to the same case as previous extension but avoid the use of activation constraints. The idea is to allow a variable to have a sub problem as a value.

The frameworks represented in [17, 22] propose two steps of resolution. The first step consists of determining the final product architecture and the second consists of resolving the corresponding classical CSP problem given the product architecture.

One of the characteristics of design activities, specially during the early stage is the existence of imprecision and uncertainty factors in the problem model and in external input [6, 21]. None of the previous frameworks offers a technique allowing a designer to take into account this uncertainties in his decision making. However, we can find in [13, 18] a probabilistic approach based on **Monte Carlo** method which is time and resource consuming.

In this paper we propose an approach to deal with uncertainty based on CSP framework which guarantees an upper bound of solution existence probability. Our discussions are focused on two points:

1. A description of the generic approach to help designer to structure the design knowledge of a product being considered; this knowledge is expressed in a generic model which is used to generate concepts;
2. A proposition of an approach to deal with uncertainty which guarantees an upper bound of solution existence probability; we will show by a case example how this measure could be an important element of decision.

[1] ONERA/DCSD, 2 av. E. Belin, BP 4025, 31055, Toulouse, France, email: {bensana,mulyanto,verfaillie}@cert.fr

As we are applying the proposed approach in aeronautical domain, examples found in this paper will be on aircraft design problems.

2 Presentation of the generic model

Our generic approach to deal with configuration problem is based on generic model of the product being designed. The generic model combines two frameworks:

- CSP framework with the notions of variables, domains and constraints to formalize the configuration problem;
- Objet-oriented representation with the notion of *objects*, *attributes*, *classes*, *instances* and *inheritance* to structure the configuration knowledge.

2.1 The configuration knowledge

Configuration knowledge, usually related to a given design domain, is structured in classes following an object-oriented representation. Here, a class can be seen as a template of an item. An instance is build upon class description corresponding to an item (final product or a product component).

2.1.1 Classes as generic design knowledge

The knowledge about items is modeled by classes. Typically there is a class for the final product and classes for its components (and recursively). A class can be:

- abstract (instances cannot be built upon directly);
- generic (*e.g. sizeable component*, instance can be built upon, its exact size is not predefined);
- non sizeable (only existing instances of this class are taken into account during the product instantiation).

Simple inheritance is considered, *i.e.* a class has only one mother-class. A class is described by a set of typed attributes.

Attributes

Each attribute is characterized by a definition (mainly its name and type). The domain of values associated with a variable depends on the attribute type. Our system allows the following types:

- `oneof(List)`: `List` is a list of elements of any discrete type (number, atom, string, *etc.*) and the value can be any element of `List`;
- `oneof(Min,Max)`: `Min` and `Max` are two integers, and the value of the variable can be any integer belonging to the `[Min, Max]` interval;
- `instance(Cl1)`: `Cl1` is a class name and the value can be any instance of any subclass `Cl2` of `Cl1` if `Cl1` is abstract or an instance of `Cl1` if not;
- `instance(List)`: `List` is a list of classes name and the value can be any instance of any class in `List`;
- `instances(Min, Max, Cl)` : `Cl` is a class name and the value can be any collection of instances of the class `Cl` whose cardinality belongs to the interval `[Min,Max]`;
- `interval(Min,Max)`: `Min` and `Max` are real numbers and the value can be any subinterval of the interval `[Min,Max]`.

The five first types introduce direct choice points since they all lead to a finite set of possibilities. The last type may also lead to choice points but only when interval splitting is considered. The inheritance property will apply every attribute of a class to its specializations.

For example, the class associated to a wing may has:

- attributes describing wing geometry in real numbers: wing $Area, Span$ and $Aspect\_ratio$, *etc.*;
- attributes determining wing components: flaps, slots, *etc.*

Generic constraints

Generic constraints are expressed at the class level. They define relationships between:

- the values of some attributes of the class; as an example, the three geometric attributes of a wing $Area, Span$ and $Aspect\_ratio$ are related by the constraint: $Aspect\_ratio = Span^2/Area$
- the values of the attributes of some of its components; for example the relation between the wing mass with the mass of its components: $wing\_total\_mass = wing\_mass + flap\_mass + slot\_mass$

In the implementation we have two types of constraints as follows:

- *pre-instantiation* constraints help to prune the search space and avoid to consider wrong options during the product instantiation; there is only a small number of constraints which can be expressed in this way depending on the structural choices for an instance;
- *post-instantiation* constraints are used to verify the instance's attributes consistency during the product instantiation.

Using the inheritance properties, all of constraints in a class are also applied to all of class specializations.

2.1.2 Instances as components

There are two types of instances:

- *generic* instances resulting directly from the instantiation of classes using the product instantiation;
- instances modeling *existing components* or Components On The Shelf (COTS) where all of the attributes have a fixed value (at least those describing its structure); these instances are recorded in a base of components.

After a product instantiation of an item represented by a generic class may result in a generic instance or a component in COTS. But, an item represented as non sizeable class can only be instantiated in one of a component in COTS. COTS can be used to perfectly represent the components non-customizable. In aircraft design, the engines are usually considered as COTS.

2.1.3 Specific constraints as designer's model

A *model* is associated to the class to be instantiated. It determines a set of basic restrictions expressed in unary constraints. The constraints defined in a model takes into account the design requirements and the designer preferences.

A model describes:

- the requirements imposed to the product or its components; in civil aircraft design domain, a requirement can be the typical aircraft flight mission, aircraft capacity, *etc.*
- explicit designer preferences, such as a simple trapezoidal wing plan-form or door type selections.

2.2 The synthesis function

The second main part of our system is the synthesis function. This function is in charge of the product instantiation *i.e.* to instantiate the class corresponding to the product using the available knowledge and product model. Here, we find the connection between CSP formalism and the object-oriented problem representation. During the instantiation of a class, a variable is associated to each attribute. An instance can be considered as a set of variables.

A synthesis function can be expressed as below:

$$Instances = synthesis(Domain, Class, Model, Comps)$$

It builds a set of instances of the class $Class$, with:

- $Domain$: the set of classes in generic knowledge;
- $Model$: a set of constraints which represent the designer preferences and design requirements;
- $Comps$: a set of re-used components in COTS.

The last two entries can be empty. When both are empty, only generic instances will be considered. Other parameters have been defined to limit the search to the first n solutions or within a time limit and to use different levels of constraint propagation.

The set of available classes and COTS implicitly describes the product instantiation search space. Techniques utilized to explore the search space in synthesis function are basically tree search and constraint propagation techniques. During the product instantiation, search in the synthesis function takes into account:

- design choices expressed in class descriptions: choice of a class among several possibles classes defined in a domain attribute, choice of a component in COTS, finite domain attributes, *etc.*
- the choice between reusing an existing component in COTS or building a new generic one;
- other choices related to the management of propagation over intervals.

Depending on the component set and the model, the same synthesis function can be used to:

- *validate* an existing concept using only non-sizeable classes; this can be interesting to verify weather a proposed configuration satisfy the product requirements;
- *define* new concepts : generic classes are authorized and the set of components is reduced to the set of COTS.

2.3 Implementation

We chose a constraint programming language to be our supporting language. The continuous domains is useful to express the sizeable properties on an item. Two languages, although not fully equivalent are selected: `Prolog IV` [4] and `Eclipse` [8]. In our observations, `Prolog IV` is more powerful than `Eclipse` in terms of constraint propagation mechanism, but is much slower as far as pure search is concerned.

The approach combining constraints and objects bears some similarities with the idea implemented in the `Claire` language [11, 7] (notions of parameterized class, abstract class, set type attribute *etc.*). But as `Claire` does not provide, presently, constraints over real intervals, this option has been discarded. In our opinion, it is easier to add basic object-oriented features adapted to our needs in an existing constraint programming language than to develop an efficient constraint propagation mechanism over an existing object oriented language.

To avoid a premature choice between any constraint programming language, we defined a kind of meta-language to express:

- generic constraint expressions;
- generic predicates defining classes and attribute;
- generic predicates accessing attribute values;
- generic predicates to control the product instantiation process.

Both constraint programming language chose do not support a graphical interface. To overcome this disadvantage, web-based utilities such as Netscape, HTML, Javascript, Perl/CGI and Wais are used.

The resulting system, still under improvements, was used for the civil aircrafts design studies [16]. It is currently used for a design of High Altitude Long Endurance Unmanned Aerial Vehicles (HALE UAVs) [5].

UAVs design is an interesting domain for configuration since an UAV can be more or less tailored for the application depending mainly on the type of sensors to be loaded and the flight mission characteristics (range, endurance, altitude, *etc.*).

The current UAV domain is organized such that different aircraft types can be considered: classical configuration plane, flying wing, airplane with a canard wing, *etc.* Different propulsion systems are described : piston engines, turbojets, electric motors with solar arrays or batteries, fuel cells as well as different types of sensors : electro-optical cameras, synthetic aperture radars, spectrometers, data transmission relays. This domain can be easily extended by filling up the domain knowledge by other classes including more general classes.

2.3.1 A small example

Let be the domain considered consists of the following classes:

```
propulsion::
 [[engine,instance(engine)],
  [nb_engines,oneof(1,2)]].

engine::
 [[consumption,interval(0.0,10.0)]].

turbomachine(engine)::
 [[thrust,interval(0.0,100.0)],
  [bypass_ratio,interval(1.0,10.0)]].

turbojet(turbomachine)::
 [[thrust,interval(0.0,10.0)],
  [bypass_ratio,1.0],
  [consumption,interval(0.0,8.0)]].

turbofan(turbomachine)::
 [[thrust,interval(0.0,50.0)],
```

```
  [consumption,interval(0.0,5.0)]].

piston_engine(engine)::
 [[power,interval(0.0,100.0)],
  [r_propeller,interval(0.7,0.9)],
  [d_propeller,interval(0.0,3.0)]].

abstract([engine, turbomachine]).
```

We suppose that the `engine` and `turbomachine` classes are abstract. `turbomachine` and `piston_engine` are classes resulting of a specializations from the class `motor` and `turbojet` and `turbofan` are specialized class from the class `turbomachine`. A specialized class is allowed to have a refined attributes domains from its motherclass as for the class `turbojet`.

Without any specific constraint, six solutions are built by the synthesis function: `s-1, s-2` with turbojet, `s-3, s-4` with turbofan engine and `s-5, s-6` with piston engine.

```
s-1::[[engine,tj-1],[nb_engines,1]]
s-2::[[engine,tj-2],[nb_engines,2]]
s-3::[[engine,tf-3],[nb_engines,1]]
s-4::[[engine,tf-4],[nb_engines,2]]
s-5::[[engine,pe-1],[nb_engines,1]]
s-6::[[engine,pe-2],[nb_engines,2]]
```

with the components:

```
{tj-1,tj-2}::[[consumption,[0.0,0.8]],
              [thrust,[0.0,10.0]]]

{tf-3,tf-4}::[[consumption,[0.0,0.5]],
             [thrust,[0.0,50.0]]]

{pe-1,pe-2}::[[consumption,[0.0,10.0]],
              [power,[0.0,100.0]],
              [r_propeller,[0.7,0.9]],
              [d_propeller,[0.0,3.0]]]
```

If we add a constraint at the `motorisation` stating that if two motors are considered, they can be only turbojets,

```
not((M.nb_engines = 2),
    ((M.engine.class = turbofan)
     ;
     (M.engine.class = piston_engine)))
```

the previous solutions `s-4` and `s-6` will not be built because they will not be consistent.

3 Dealing with uncertainty

After presenting our system in the previous section, in this section we describe our approach to deal with uncertainty in conceptual design, the earliest phase in design activity. We based our proposed approach on CSP framework. We begin by observing two kinds of variable natures.

While representing a design problem as a constraint satisfaction or constraint optimization problem, all the variables do not represent the same thing. Some of them represent possible designer choices. One says that they are controllable. Some others represent uncertainty or imprecision. One says that they are uncontrollable. But such a distinction does not exist in the standard CSP framework. Both are dealt with the same way.

There are at least two extensions of the CSP framework that aim at dealing with uncertainty:

- **Probabilistic Valued-CSP** [20] associates with each constraint (or each combination of values) a probability of existence in the real world (or probability to be forbidden in the real world). The objective is then to find an assignment for all the variables that maximizes its probability to be solution in the real world;
- **Mixed-CSP** [9] explicitly distinguishes controllable and uncontrollable variables. The objective is then to find an assignment for all the controllable variables that is a solution whatever the assignment of the uncontrollable variables is. But, if a probability distribution is available on the domains of all the uncontrollable variables, the objective can be, as in the Probabilistic Valued CSP framework, to find an assignment for all the controllable variables that maximizes its probability to be solution in the real world.

Both extensions only differ in the way of expressing uncertainty: in the first framework, uncertainty is associated with constraints or combinations of values, while variables and domains are the same as in the standard CSP framework; in the second framework, uncertainty is associated with values of uncontrollable variables, while constraints are the same as in the standard CSP framework. At least theoretically, any problem expressed in one of the above frameworks can be expressed in the other.

To deal with uncertainty in design problems, we choose the second framework based on the distinction between controllable and uncontrollable variables. Using the generic model presented previously:

1. we express this distinction in the definition of the classes; then any variable involved in a design problem is either controllable, or not; it suffice to state weather an attribute is controllable or not;
2. we use this distinction to provide the designer with useful information about the consistency probability of the current problem.

We will see that such an extension does not imply any change neither in the generic model, nor in the synthesis function presented above.

3.1 Controllable variables

Here, controllable means that the assignment of a value to the variable is under the control of the designer. We can distinguish three types of controllable variables:

- design variables: they physically describe the designed product; in the example of section 3.4, the attributes representing wing span, the wing area, the lift coefficient, and the aircraft drag due to surface friction are all design variables;
- evaluation variables: they characterize the performance of the designed product; in the same example, the attribute lift-to-drag ratio is an evaluation variable;
- intermediate variables: they are used to simplify the problem expression; still in the same example, the aspect ratio AR is an intermediate variable.

3.2 Uncontrollable variables

On the contrary, uncontrollable means that the assignment of a value to the variable is not under the control of the designer. This may be due to the presence of:

- imprecision or uncertainty in the available design knowledge *i.e.* in item model expressed in classes;
- imperfectly known environment factors; some inputs maybe imprecise or uncertain;
- other designer decisions in a distributed design context.

3.3 Proposed approach

We make the following assumptions:

- uncertainty can be modeled as a probability distribution on the domain of each uncontrollable variable;
- all the uncontrollable variables can be considered as independent.

Using the constraint propagation mechanism, during the product instantiation, controllable variable assignments are propagated in the whole problem and may induce domain reductions on the uncontrollable variables. The smaller the size of the current domains of the uncontrollable variables, the smaller the probability of existence of a real solution in the current problem. We simply propose to use the size of the current domains of the uncontrollable variables to compute an upper-bound on the probability of existence of a real solution in the current problem.

More formally, a $CSP = (V, D, Pr, C, \alpha)$ is defined:

- $V = (Vc \cup Vu)$ where $Vc = \{vc_1, ..., vc_n\}$ is the set of controllable variables, and $Vu = \{vu_1, ..., vu_m\}$ is the set of uncontrollable variables ;
- $D = (Dc \cup Du)$ where $Dc = dc_1 \times ... \times dc_n$, dc_i is the domain associated with vc_i, $Du = du_1 \times ... \times du_m$, and du_j is the domain associated with vu_j;
- $Pr = \{\pi_1, ..., \pi_m\}$ where π_j is, for each uncontrollable variable, the probability distribution associated with du_j;
- C a set of constraints, each of them involving at least one controllable variable;
- $\alpha = \prod_{j=1}^{m} \alpha_j : \alpha_j = f(du_j, \pi_j)$, where $f(du_j, \pi_j) = \int_{du_j} \pi_j(z).dz$ for continuous domains, and $f(du_j, \pi_j) = \sum_{z \in du_j} \pi_j(z)$ for discrete domains.

Under the independence assumption, α is an upper-bound on the probability of existence of a real solution in the current problem. If we make the following assumptions:

- for each uncontrollable variable, the probability to take its value in its initial domain (before synthesis, *i.e.* before constraint propagation and tree search) is equal to 1;
- for each uncontrollable variable, the associated probability distribution is uniform.

α_i is simply the ratio between the size of the current domain of λ_i and the size of its initial domain. α is consequently very easy to compute at any step of the synthesis.

Moreover, by associating each uncontrollable variable with one item (product or component), such an approach can be easily integrated in the object-oriented framework.

3.4 A problem example

We take an example of UAV design for cruising flight. The designer has to determine a wing placement with regard on aircraft body that has good aerodynamic properties by considering the wing geometry uncertainty. In this example the wing placement is represented by the lift coefficient variable.

Here, we consider two sizeable components : aircraft component and wing component, where the wing component is a subpart of aircraft component. Below, we define the `wing` and `aircraft` classes.

The wing class has the following attributes:

- wing span: `span`
- wing surface: `area`
- wing Oswald factor: `oswald`

The Oswald factor is a parameter representing the lift distribution along the wing. It is affected by many other specific wing parameters such as twist angle distribution and wing airfoil distribution. In the early stages of design, this information is not available. One can consider the Oswald factor as an uncontrollable variable taking its value over a given interval. In this example, we take an interval between 0.7 and 0.8. There is no constraint in the `wing` class definition.

The `aircraft` class has the following attributes:

- aircraft aerodynamic property represented by the lift-to-drag ratio: `lift_to_drag`
- aircraft lift coefficient: `c_lift`
- aircraft total drag coefficient: `c_drag`
- aircraft surface friction drag coefficient: `c_drag_fr`
- aircraft wing component: `ac_wing`

The `aircraft` class contains constraints between these variables and the variables in the `wing` class. In this example we assume that the surface friction drag coefficient does not depend on the wing geometry variation.

```
wing::
 [[span, interval(0.0,10.0)],
  [area, interval(0.0,20.0)],
  [unc(oswald), interval(0.7,0.8)],
  [alpha, interval(0.0,1.0)]].

aircraft::
 [[lift_to_drag, interval(0.0,20.0)],
  [c_lift, interval(0.0,1.0)],
  [c_drag, interval(0.0,1.0)],
  [c_drag_fr, interval(0.0,1.0)],
  [ac_wing, instance(wing)],
  [alpha, interval(0.0,1.0)]].
```

The predicate `unc(X)` defines `X` as an uncontrollable variable. The following constraints are expressed at the `aircraft` level:

```
aircraft_constraints(Aircraft)
 {
  Wing = Aircraft.ac_wing
  Sp_2 = power(Wing.span, 2.0)
  AR   = div(Sp_2, Wing.area)
  Cl_2 = power(Wing.c_lift, 2.0)
  E    = Wing.oswald
  K    = product(pi, AR, E)
  CDi  = div(Cl_2, K)
  Wing.c_drag = plus(Wing.c_drag_fr, CDi)
  }
```

`Aircraft` is a variable representing the current instance being built, `power`, `div`, `plus`, `product` belong to the set of basic constraints defined in the language and `pi` refers to the value of π.

Let us assume now that the designer has set some controllable variables and expressed his preferences (in IU metrics) in the following aircraft model:

```
Aircraft.ac_wing.span = 6
Aircraft.ac_wing.area = 4.2
Aircraft.c_drag_fr    = 0.02
Aircraft.c_drag      <= 0.0285
```

Let us suppose that the designer has two design alternatives to compare:

1. `Aircraft.c_lift = 0.4`
2. `Aircraft.c_lift = 0.42`

After running the synthesis function, we find the following results for the lift-to-drag ratio, the Oswald factor and the α parameter:

First alternative:

```
Aircraft.lift_to_drag   in [14.04..14.59]
Aircraft.ac_wing.oswald in [0.7..0.8]
Aircraft.alpha           = 1.0
```

Second alternative:

```
Aircraft.lift_to_drag   in [14.73..14.90]
Aircraft.ac_wing.oswald in [0.77..0.8]
Aircraft.alpha           = 0.3
```

An aircraft has a better aerodynamic property when it has higher lift-to-drag ratio. The designer may then choose the second alternative because of its higher lift-to-drag value, but this solution induces a lower value of the upper-bound on the probability of existence of a solution, *i.e.* a higher risk. Thus, a prudent designer may choose the first alternative instead of the second.

4 Conclusion and perspectives

We have presented a generic approach for configuration and design which associates object and constraint programming technologies. The basic scheme has been extended to deal with some of the uncertainties a designer may encounter in the earliest stages of design.

Possible future developments could consider the introduction of dynamic parameters. Such parameters may also support probabilistic aspects like the probability that a given characteristic takes a given value at a given point of time. Modeling such knowledge will help to address prospective design and give answers to questions such as: is it possible to design a product with a given level of performance within a given time horizon.

Other developments could consider the "compilation" of the knowledge base. Once the domain description is stable, the actual synthesis could be replaced by a more efficient version, by "compiling" the set of classes and producing a constraint program where all design choices are for example expressed as discrete domain variables. This would allow the use of more powerful propagation techniques and helps to define specific design domain configurators.

REFERENCES

[1] T. Barret, G. Coen, J. Hirsh, L. Obrst, J. Spering, and A. Trainer, 'Madesmart : an integrated design environment', in *ASME Design for Manufacturing Symposium*, (Septembre 1997).

[2] C. Barrouil, E. Bensana, C. Castel, L. Chaudron, C. Cossart, R. Manpey, and C. Tessier, 'Programme perception', Rapport final 96-97 RF 2/7996.34 DCSD-T, ONERA/DCSD, (Mars 1998).

[3] G. Bel, M. Barat, and C. Gœuriot, 'Programme PERCEPTION : thème Conception et Dimensionnement', Rapport final 2/7995.02-3575.00, CERT/DERA, (Octobre 1996).

[4] F. Benhamou, 'Lecture notes in computer science', in *Constraint Programming : basics and trends*, ed., A. Poldelski, volume 910 of *Lecture notes in computer science*, chapter Interval constraint logic programming, 1–21, Springer Verlag, (1994).

[5] E. Bensana, 'Etudes conceptuelles d'avions non pilotes : apports de la programmation par contraintes sur les intervalles', in *2eme Congres de la Société Francaise de Recherche Opérationnelle et d'Aide la Décisiosn*, (Janvier 1999).

[6] D.C. Brown, 'Design', in *Encyclopedia of Artificial Intelligence Vol 1*, ed., S.C. Saphiro, 331–339, John Wiley and Sons, New York, (1992).

[7] Y. Caseau and F. Laburthe, 'Introduction to the CLAIRE programming language', Technical report, Département Mathématiques et Informatoique, Ecole Normale Supérieure, France, (1997).

[8] ECRC. Eclipse related papers and reports. `http://www.ecrc.de/eclipse/html/reports.html`.

[9] H. Fargier, J. Lang, and T. Schiex, 'Mixed Constraint Satisfaction : A Framework for Decision Problems under Incomplete Knowledge', in *AAAI-96*, pp. 175–180. AAAI Press, (1996).

[10] I. Kroo, S. Altus, R. Braun, P. Gage, and I. Sobelski, 'Multidisciplanary optimization methods for aircraft preliminary design', *AAIA*, **94**(4325), (1994).

[11] F. Laburthe and Y. Caseau, 'Ecrire du code élégant pour des algorithmes complexes', *Langaes et Modèles à Objets*, (1996).

[12] A.K. Mackworth, 'Consistency in Networks of Relations', *Artificial Intelligence*, **8**(1), 99–118, (1977).

[13] D.N. Mavris, O. Bandte, and D.A. DeLaurentis, 'Robust Design Simulation : A Probabilistic Approach to Multidisciplinary Design', *Journal of Aircraft*, **36**(1), 298–307, (jan 1999).

[14] S. Mittal and B. Falkenhainer, 'Dynamic Constraint Satisfaction Problems', in *8th National Conference of Artificial Intelligence*, pp. 25–32, Boston, MA, (1990). AAAI-90.

[15] F. Morel, 'Multivariate optimization applied to conceptual design of high capacity log range aircraft', *AIAA*, (1994).

[16] T Mulyanto, *Utilisation d'outil de programmation par contraintes pour l'étape conceptuelle de la conception d'avions*, Master's thesis, ENSAE, Aout 1998.

[17] B. O'Sullivan and J. Bowen, 'A Constraint-based Approach to Supporting Conceptual Design', in *Artificial Intelligence in Design '98*, pp. 291–308, Instituto Superior T cnico, Lisbon, (jul 1998).

[18] B. Roth, D. Mavris, and D. Elliott, 'A Probabilistic Approach to UCAV Engine Sizing', in *Joint Propulsion Conference, AIAA98-3264*, Cleveland, OH, (1998). AIAA.

[19] D. Sabin and E.C. Freuder, 'Configuration as Composite Constraint Satisfaction', in *AAAI-96 Fall Symposium on Configuration*, pp. 28–36, (1996).

[20] T. Schiex, H. Fargier, and G. Verfaillie, 'Problème de satisfaction de contraintes valué', *Revue d'Intelligence Artificielle*, **11**(3), 339–373, (1997).

[21] T.W. Simpson, D. Rosen, J.K. Allen, and F. Mistree, 'Metrics for Assessing Design Freedom and Information Certainty in the Early Stages of Design', *Journal of Mechanical Design*, **120**, 628–635, (dec 1998). Transactions of the ASME.

[22] M. Veron, H. Fargier, and M. Aldanondo, 'From CSP to Configuration Problems', in *AAAI-99 Workshop on Configuration*, Orlando, Florida, (jul 1999).

Preference-based Configuration of Web Page Content

Carmel Domshlak **Samir Genaim** **Ronen Brafman** [1]

Abstract. In this paper we present a new approach for personalized presentation of web-page content. The model is defined as a preference-based configuration process that is based on qualitative decision theory. This configuration process attempts to determine the optimal presentation of a web page while taking into account the preferences of the web author as well as viewer interaction with the browser. The preferences of the web-page author are represented by a CP-network, a graphical model developed in [2]. We present an implementation of our approach in a CPML system and discuss the implementation issues.

1 INTRODUCTION

An important goal for web-page designers is the ability to provide viewer oriented personalization of web-page content. Two approaches to this problem are currently in use: (1) Learning user profiles; (2) Online information gathering by shared keywords. Both approaches are useful in particular applications, but as general solutions they have some important drawbacks: The first approach addresses only long-term user preferences, and therefore, it is only applicable to frequent viewers. In addition, it reacts slowly to shifts in user interests. The second approach uses keywords that show up in the material the user requested as a basis for fetching and presenting additional information that may also interest the user. This approach is more dynamic in that it reacts immediately to changes in the user's interests. However, the web-page designer has little, if any, control over the content of different web-page components.

In this work we propose a new model for representing the content of a web-page. This model reflects the preferences of the web-page author while adjusting dynamically to the viewer's current interests. First the web-page author expresses his expectations regarding the presentation of the web-page content. For example, the author may prefer some material to be fully presented if and only if the heading of some other material is presented. This is done in an intuitive yet expressive manner. These expectations become a *static* part of the web document, and set the parameters of its initial presentation. For any particular user, the actual presentation then changes *dynamically* to accommodate that user's choices and actions. We achieve content personalization through dynamic preference-based reconfiguration of the web page. Note that a web page is usually composed of a collection of components, and the information content of each component can be presented in different ways. For example, an article can be presented by its heading, summary, by partial content, or by the full content. The content's configuration process attempts to determine the best presentation of all web page components with respect to the web-page author's preferences and the viewer's behavior.

Our approach is based on qualitative decision theory, and we argue that it provides appropriate support for web page configuration. We use the CP-network model [2] to represent the web-page author's preferences. A CP-network is a qualitative, graphical model of preferences, that captures statements of conditional preferential independence. We implement this approach in the framework of the CPML system, which consist of the authoring tool for the preference-based web pages, and a corresponding viewing tool, which is implemented as a browser plugin.

The paper is organized as follows: In Section 2, we present the framework of the web page preference-based configuration in the context of qualitative decision making. We then discuss the relevant preference representation issues, and describe the CP-network model. In Section 3, we describe the architecture and implementation of the CPML system. In Section 4, we summarize the presentation and discuss future work.

2 PREFERENCE-BASED WEB PAGE CONTENT PERSONALIZATION

In this section we present preference-based web page configuration, and show how decision theoretic tools provide a basis for this application. General preference-based configuration was previously investigated [1, 3, 4], however its adaptation to information personalization has not been explored.

In preference-based configuration, a decision maker chooses a number of different components, that together form the final product. The chosen configuration should be the best (or at least an undominated) configuration with respect to the decision maker's subjective preferences. The whole process of a partly unsupervised, preference-based configuration can be divided into three stages:

1. an *identification step*, which consists of selecting relevant criteria,
2. a *specification step*, which consists of interactive elicitation of user preferences,
3. a *processing step*, which consists of reasoning about component selection with respect to the user's preferences,

Note that the user alluded to in stages 2 and 3 is the web-page designer. Here we describe a preference-based configuration of the web page content. We first provide a formalization of this problem, then we discuss the preference representation issues.

2.1 Configuration and qualitative decision theory

A web page consist of a finite set of *components* $C = C_1, \ldots, C_m$, each one with an associated finite domain of *component values*, $\mathcal{C}_i = \{c_1^i, \ldots, c_{N_i}^i\}$. For example, the components can be articles,

[1] All authors are from Department of Computer Science, Ben Gurion University of the Negev, P. O. Box 653, Beer-Sheva 84105, Israel, e-mail: {dcarmel,brafman,genaim} @cs.bgu.ac.il

pictures, commercials, etc., and each component may have different possible presentations (= values), such as full content, partial content, title, link, invisible, etc. The *configuration space* for the web page content is the set $\mathcal{C} = \mathcal{C}_1 \times \cdots \times \mathcal{C}_m$, and each element $c_k \in \mathcal{C}$ is a possible content *configuration*.

In order to determine the best configuration we define an order over the configuration space. A preference ranking is a total preorder $\succeq$ over the set of configurations: $c_1 \succeq c_2$ means that configuration c_1 is equally or more preferred to the decision maker than c_2. Of course, the ordering $\succeq$ will be different for different decision makers. Given a preference order $\succeq$ over the configuration space, an *optimal configuration* is any $c \in \mathcal{C}$ such that $c \succeq c'$ for any $c' \in \mathcal{C}$.

A classical preference-based configuration of a product is usually driven by the subjective preferences of the actual consumer. In our application the role of the decision maker is given to another actor. During the design of the web page, the web-page author describes her expectations regarding content presentation. This way, once the web-page author defines the preference for a web-page, the preference elicitation proccess is done dynamicly each time this pages is accessed, i.e the preference changed depending on the current interest of the user. *The preference order $\succeq$ represents the subjective preferences of the web page author, not of its viewer.* However, during the dynamic configuration of the web page content, the actual choices of the viewer affecting the configuration decisions.

The choice of a preference model is a central design decision in any preference-based configuration process. In particular, one can adopt a quantitative, utility-theoretic, model, or a more qualitative model. We believe that qualitative models[2] can form a good basis for the automated product configuration process in general, and for the web page configuration process in particular. The main advantages of the tools of qualitative decision theory, relatively to the tools of traditional decision theory, are compactness, intuitiveness, and reduced computational effort. Two major reasons motivate us towards qualitative decision making in our domain: Utility assessments are likely to be unintuitive in our setting. However, the web-page designer is likely to be able to compare and to rank alternative designs. In particular, the designer is likely to find it natural to specify different alternative representations for a particular web-page component given a fixed setting. Moreover, since the number of options for representing each component is rather small, ordering different options for each component is likely to be a relatively easy task.

2.2 Preference representation

Although a qualitative representation of preferences is typically simpler to represent and manipulate, the preference elicitation stage can still be quite complicated. To perform real-life preference-based configuration, we must represent user preferences in a compact, yet expressive manner. Even relatively small web page may consist of many different components, with varying importance and internal relations. Therefore we cannot afford to handle explicitly a ranking table for all the alternatives. Likewise, we want to capture conditional preference dependencies between different components. For example, the web-page author may prefer to present a Volvo commercial if the user explicitly examined the content of an article about a traffic accident, but to present a Linux commercial otherwise.

An appropriate representation of qualitative preferences is insufficient on its own; we need to be able to reason efficiently about these preferences. Possible reasoning tasks include finding the optimal configuration, comparing two possible configurations, etc. Likewise, the representation model should be intuitive and natural in order to simplify and facilitate the web-page designer's task.

Because of these requirements, in our work we decided to exploit the advantages of the CP-network model developed by Boutilier *et al.* [2]. This is an intuitive, qualitative, graphical model of preferences, that captures statements of conditional preferential independence. Therefore it provides an appropriate basis for modeling preferences of web authors. In our domain, each node in the network (a directed acyclic graph – DAG) is identified with a web page component C_i. The immediate parents of C_i, $\Pi(C_i)$ are components that affect preference of the web-page author over the possible presentations of C_i. Formally, if $\overline{C_i} = C \setminus \{C_i, \Pi(C_i)\}$, then C_i and $\overline{C_i}$ are conditionally preferentially independent given $\Pi(C_i)$. Each node C_i contains a table (CPT), which describes the web-page author's preferences about the values of C_i given all possible combinations of $\Pi(C_i)$. The construction of a CP-network for C consists of two stages: for each component C_i, asking the web-page author to identify $\Pi(C_i)$, and to specify $CPT(C_i)$. Each complete assignment for the vertices of the network represents a configuration of the web page content. Figure 1 shows an example of a CP-network with the corresponding CPTs. Determining if a configuration $\mathbf{x}$ is preferred to another configuration $\mathbf{y}$ can be defined as inferring whether $\mathbf{x} \succ \mathbf{y}$ is a consequence of a given CP-network. For some domains this task requires low polynomial time [2, 5]. Likewise, a branch and bound algorithm for finding best feasible configuration was presented by Boutilier *et al.* in [1].

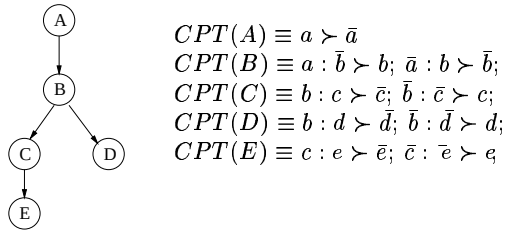


Figure 1. An example CP-network

As an example consider an online newspaper web page which is divided into the following three areas: an area for a recently requested article, an area for the article summaries, and an area for links (titles) to the remaining articles. Suppose that today's newspaper consist of five articles. The author defines the preferential relations between them by the CP-network presented in Figure 1. For example, the value b of the article B corresponds to its content (summarized or full, depending on the viewer's recent behavior), while the value $\bar{b}$ corresponds to the link to this article. The preference of the web-page author on whether to introduce or to hide the content of the article B depends on the actual presentation of the article A. Namely, if the content of A is presented, then the web-page author does not want to focus the attention of the viewer on article B, and hence, its content should be hidden. Otherwise the author prefers to expose it. Handling a nested structure also possible if we add more nodes to the CP network , or if we allow non-binary values, which makes the network more complex.

In the following section we describe the implementation issues of the CPML system for preference-based configuration of the web

[2] A comprehensive overview of the field of qualitative decision theory can be found in [6].

pages.

3 CPML SYSTEM DESCRIPTION

CPML is a prototype system currently being developed at Ben-Gurion University for preference-based authoring and viewing of interactive web pages. The outline of the system is presented in Figure 2. The system can be divided into two modules - the authoring tool, and the viewing tool.

The authoring tool allows web-page authors both to create a web page, and to define preferences on the values of its components. The first task is standard for web-page authoring tools. However, in CPML it is adapted to make subsequent definition of preferences easy to the web-page author. The content of the web pages is described in dynamic HTML (DHTML) [7]. It consist of HTML blocks, which are wrapped by javascripts, to allow presenting their content in different manners. The second task is defining a CP-network that captures the preferences of the web-page author on the configurations of the web page. The editor for CP-networks is integrated in the authoring tool, and tightly coupled with the web page creation process. During the creation of the web page, all its components are tagged, thus can be declared as variables in a corresponding CP-network.

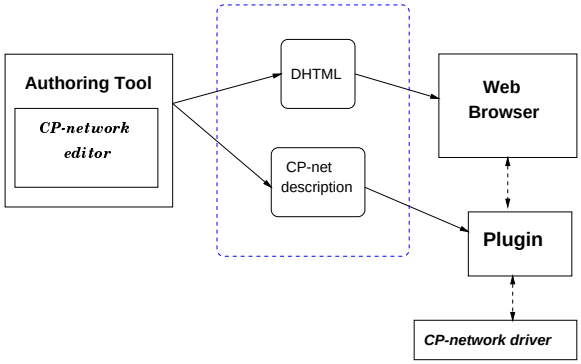


Figure 2. Framework of the CPML system

The client side consists of a standard browser expanded by a CPML agent, implemented by a browser-compatible plugin. Accessing a web document in our system results in shipping the document to the browser and the embedded CP-network to the agent. On the download of a preference-based web document, the agent sets all the components to their corresponding values in the optimal configuration, given no evidences on the viewer's current interests. From this point the agent acts in event-driven fashion, and waits for a viewer interaction with the browser (exposing/hiding a component). The agent is responsible to react to the viewer's actions by changing the content presentation of the document. The change is performed according to the embedded CP-network and the recent actions of the viewer. The best configuration of the web page, given the recent choices of the viewer, is determined, and the web page is redrawn.

Determining the best configuration is done by the procedure AgentResponse, presented in Figure 3. At this current stage of the system development we restrict ourself to components with only binary domains in order to exploit significant computational benefits, described in details in [5]. Therefore, in step 1 of AgentResponse, we *switch* the value of the observed component, and then add it to the list

Procedure AgentResponse(C_i)
var:
$\mathcal{G}$ - CP-network of the web page's components
$\mathcal{E}$ - List of the events (component, value) produced by the viewer

1. Switch the value of C_i and add it to the chronologically sorted event list $\mathcal{E}$. If $\mathcal{E}$ is bigger than a specified threshold: remove the oldest event from $\mathcal{E}$.
2. Set c to the optimal configuration returned by BestConfiguration $(\mathcal{E}, \mathcal{G})$

Procedure BestConfiguration $(\mathcal{E}, \mathcal{G})$

1. Reduce graph $\mathcal{G}$ to $\mathcal{G}_\mathcal{E}$ by projection of $\mathcal{E}$ on it, and let $\mathcal{G}^1_\mathcal{E}, \mathcal{G}^2_\mathcal{E}, \ldots, \mathcal{G}^m_\mathcal{E}$ be the connected components of $\mathcal{G}_\mathcal{E}$
2. For each $\mathcal{G}^i_\mathcal{E}$, and for each variable $v \in \mathcal{G}^i_\mathcal{E}$ (according to a topological sort) set v to its most preferred value with respect to $\Pi(v)$ and $\mathcal{E}$

Figure 3. Preference-based reactive configurator

$\mathcal{E}$ of the recent viewer's choices. The maximal size of $\mathcal{E}$ is specified by the author of the web page and is passed as a part of the document. Subsequently called procedure BestConfiguration performs the optimization and returns one of the pareto optimal configurations, consistent with the preferences of the web-page author ($\mathcal{G}$), and the recent choices of the viewer ($\mathcal{E}$). Due to the semantics of the CP-network model, this process can be performed in time linear in the number of components. For details about the optimization procedure consult [2].

4 SUMMARY AND FUTURE WORK

In this paper we presented a framework for preference-based configuration of the web page content. Our approach is based on qualitative decision theory, and in particular on the CP-network graphical model for preference representation. The configuration process is performed according to the preferences of the web author as well as to the current interests of the viewer. Preferences of the web-page author are encoded as a CP-network. Given this CP-network and the recent actions of the viewer, the optimal configuration of the web page content is determined.

A prototype implementation of the CPML system is currently being developed, and we plan to complete it by August 2000. It consist of an authoring tool on the web-page author's side, and the decision making agent on the client side, implemented as a browser plugin. In addition, we are implementing an overall solution that is based on Java applets instead of plugin.

REFERENCES

[1] C. Boutilier, R. Brafman, C. Geib, and D. Poole, 'A Constraint-Based Approach to Preference Elicitation and Decision Making', in *AAAI Spring Symposium on Qualitative Decision Theory*, Stanford, (1997).
[2] C. Boutilier, R. Brafman, H. Hoos, and D. Poole, 'Reasoning with Conditional Ceteris Paribus Preference Statements', in *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–99)*, (1999).
[3] Joseph D'Ambrosio and William Birmingham, 'Preference-Directed Design', *Journal of Artificial Intelligence in Engineering Design, Analysis, and Manufacturing*, **9**, 219–230, (1995).

[4] D.L.Thurston, 'A Formal Method for Subjective Design Evaluation with Multiple Attributes', *Research in Engeneering Design*, **3**, 105–122, (1991).
[5] Carmel Domshlak and Ronen Brafman, 'Structure and Complexity in Planning with Unary Operators'. (submitted to 14th Workshop "New Results in Planning, Scheduling and Design", ECAI-2000).
[6] J. Doyle and R.H. Thomason, 'Background to Qualitative Decision Theory', *AI Magazine*, **20**(2), 55–68, (1999).
[7] Jeff Rule and Jeffrey S. Rule, *Dynamic Html: The Html Developer's Guide*, Addison-Wesley, November 1998.

Exploiting structural abstractions for consistency based diagnosis of large configurator knowledge bases

Alexander Felfernig[1], **Gerhard E. Friedrich**[1], **Dietmar Jannach**[1] and **Markus Stumptner**[2]

Abstract. Debugging, validation, and maintenance of configurator knowledge bases are important tasks for the successful deployment of product configuration systems, due to frequent changes (e.g., new component types, new regulations) in the configurable products. Model based diagnosis techniques have shown to be a promising approach to support the test engineer in identifying faulty parts in declarative knowledge bases. Given positive (existing configurations) and negative test cases, explanations for the unexpected behavior of the configuration systems can be calculated using a consistency based approach.

For the case of large and complex knowledge bases, we show how the usage of hierarchical abstractions can reduce the computation times for the explanations and in addition gives the possibility to iteratively and interactively refine diagnoses from abstract to more detailed levels. Starting from a logical definition of configuration and diagnosis of knowledge bases, we show how a basic diagnostic algorithm can be extended to support hierarchical abstractions in the configuration domain. Finally, experimental results from a prototypical implementation using an industrial constraint based configurator library are presented.

1 INTRODUCTION

Knowledge based product configuration systems are a successful application of AI technology and will still gain further importance due to the fact that more and more companies start to offer their products tailored to their specific customer's needs [4]. Many approaches for efficient problem solving and knowledge representation for configuration problems have been proposed, starting from rule-based systems [1] up to higher forms of representation and reasoning techniques, e.g., constraint based, case based, or functional reasoning ([14],[18]). However, the facts that real-world configuration knowledge bases tend to get large and complex and that the knowledge bases are subject to frequent changes (e.g., new component types, new regulations and restrictions) make the validation, maintenance, and debugging activities to important tasks during the lifecycle of the configurator. In fact, especially the maintenance problem was one of the most important problems already in the first industrial configuration systems [1] where changes of up to fourty percent of the knowlege base per year are not unusual.

Techniques from model based diagnosis (MBD) which were initially applied to diagnose faults in electronic circuits and other hardware devices have shown to be also applicable for the domain of software debugging, e.g., diagnosis of logic programs [5], repairing inconsistencies in databases [12], or VHDL programs [10].

In [7], it was shown, how these techniques can also be applied for error detection within knowledge based configuration systems. Speaking in MBD terms, a system is composed from a set of components with some predefined behavior. A diagnosis is then a set of components from the system, which if considered to be faulty, can explain discrepancies between the expected behavior of the system and the actual observations. When employing MBD to debug faulty knowledge bases, the role of the system's components is taken by the elements of the knowledge base (typically logical sentences or constraints). When provided some examples (test runs, observations) the diagnostic task is to identify these faulty parts from the knowledge base which explain an unexpected behavior of the configurator. For these task, *positive* and *negative* examples can be employed. Positive examples are (partial or complete) configurations, which should be accepted by the configurator or can be extended to a complete configuration. These positive examples can be former (supposedly valid) configurations, which should still be valid after some changes in the knowledge base. Negative examples are configurations which should be rejected by the configurator (e.g., because these constellations should not be available anymore). The behavior of the configurator is said to be *unexpected* if a positive example is rejected or if a negative example is falsely accepted.

However, when debugging large and complex knowledge bases, this diagnosis technique can be costly in terms of computation time, especially in cases, when multiple faults of higher cardinality should be detected and when the explanation facilities of the employed inference engine (e.g., a specialized constraint solver) are limited. The usage of *abstraction hierarchies* with different levels of detail has been applied successfully to increase the efficiency of model based diagnosis [16]. In this paper we want to show how these abstraction mechanisms can be employed for the diagnosis of configurator knowledge bases, where we can take advantage of the typical structure of these knowledge bases. Accordingly, no additional or artificial hierarchies must be defined to allow for a hierarchical approach, where we can start the diagnosis with at a high level (with a coarse granularity) which can be calculated very fast, and can then iteratively refine those diagnoses to a more detailed level.

The paper is organized as follows: After giving a motivating example (Section 2) we shortly review the definitions of consistency based diagnosis of configurator knowledge bases from [7] and show how a hierarchical extension can improve the efficiency of the algorithm (Section 3). After presenting experimental results using a simple algorithm with an indus-

[1] Institut für Wirtschaftsinformatik und Anwendungssysteme, University of Klagenfurt, 9020 Klagenfurt, Austria; email: {felfernig,friedrich,jannach}@ifi.uni-klu.ac.at

[2] Institut für Informationssysteme, Abteilung für Datenbanken und Expertensysteme, Paniglgasse 16, A-1040 Wien; email: mst@dbai.tuwien.ac.at

trial strength configurator library (Section 4) we discuss related work and conclusions.

2 MOTIVATING EXAMPLE

For the demonstration of our approach we use a small part of a configuration knowledge base from the domain of configurable personal computers. We employ first order logic as a representation mechanism to provide clear and precise semantics. As a conceptual model, a component-port based approach is chosen [15], the graphical depiction follows the approach from [6].

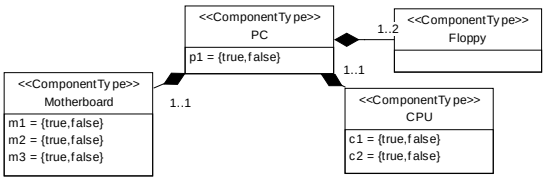


Figure 1 Example problem

We employ the following predicates for our logical representation: *types*, *ports*, *attributes*, and *dom* are functions to describe the component types of the domain with their connection points and attributes with possible values (domain). The facts *type/2*, *conn/4*, and *val/3* describe individual component type instances, their connections and the attribute valuations. We also use a logic programming notation, where variable names start with upper-case letters.

The knowledge base consists of the following definitions:

types={pc,motherboard, cpu, floppy}
ports(pc)={motherboard,cpu,floppy-1,floppy-2}.
ports(motherboard)={pc}.
ports(cpu)={pc}. ports(floppy)={pc}.
attributes(pc)={p1}. attributes(cpu)={c1,c2}. ...
dom(pc,p1)={true,false}. dom(pc,p2)={true,false} ....

In addition, the following constraints on attribute combinations of *PC*, *Motherboard*, and *CPU* have to hold:

Constraint PC_1: $(p1 = m1 \vee c1)$

$type(P,pc) \wedge type(M,motherboard) \wedge type(C,cpu) \wedge conn(P,motherboard,M,pc) \wedge conn(P,cpu,C,pc) \wedge val(P,p1,X) \wedge val(M,m1,Y) \wedge val(C,c1,Z) \Rightarrow X = Y \vee Z.$

Constraint MB_1: $(m1 = m2 \wedge m3)$

$type(M,motherboard) \wedge val(M,m1,M1) \wedge val(M,m2,M2) \wedge val(M,m3,M3) \Rightarrow M1 = M2 \wedge M3.$

Constraint MB_2: $(m2 = m3)$

$type(M,motherboard) \wedge val(M,m2,M2) \wedge val(M,m3,M3) \Rightarrow M2 = M3.$

Constraint CPU_1: $(c1 = c2)$

$type(C,cpu) \wedge val(C,c1,C1) \wedge val(C,c2,C2) \Rightarrow C1 = C2.$

Note that additional constraints may be defined for the component types (e.g., for the *floppy* type), which are omitted in the example because they are not part of any minimal conflict set.

Let us assume, that constraint PC_1 was newly introduced to the knowledge base and contains an error and should be

$$"p1 = \neg (m1 \vee c1)"$$

If we consider the only positive example to be a configuration with connected instances of one *PC*, one *Motherboard*, and one *CPU*, where *p1=false*, *m2=true*, and *c2=true*, more formally,

CONF = {type(pc1,pc). type(cpu1,cpu).
type(mb1,motherboard}.
conn(pc1,motherboard, mb1,pc).
conn(pc1,cpu,cpu1,pc) .val(pc1,p1,false).
val(mb1,m2,true). val(cpu1,c2,true).},

this partial example cannot be completed to a working configuration. The constraints $\{PC_1,MB_1,MB_2,CPU_1\}$ cause a contradiction with the positive examples. When applying the diagnosis algorithm from [7], the following minimal diagnoses are returned expressed using *ab* (abnormal) literals.

$$ab(PC_1). \quad ab(MB_1) \wedge ab(CPU_1). \quad ab(MB_2) \wedge ab(CPU_1).$$

In other words, if we consider- for each diagnosis - the individual constraints to be faulty, i.e., abnormal (and ignore it in the search for solutions) a solution to the problem can be found. The resulting diagnoses serve as pointers for the test engineer on which constraints he/she should focus the debugging efforts. (For the treatment of negative examples, see [7]).

For the calculation of diagnoses, the notion of *conflict sets* is used. A *conflict set* can be seen as a set of constraints in our knowledge base which, together with the domain description and an example, causes a contradiction. A conflict set is *minimal*, if no subset of the conflict set is itself a conflict set. Although the diagnostic algorithm does not rely on the calculation of minimal conflict sets, the size of the conflict sets heavily influences the overall efficiency of the task. In cases, where we do not have minimal conflict sets, the diagnostic algorithm tries to find explanations that contain constraints which are definitely not relevant (e.g., some constraints on the floppy component type in our example.) The calculation of minimal conflict sets can also be costly, if the employed inference engine has only limited explanation facilities.

In the following we describe an approach, where we use *structural abstraction*, i.e., the system and its components is decomposed into a hierarchy of its subcomponents. The diagnosis process is started at an abstract level where the model is usually very simple and consists of only a few high level components. The calculation of solutions can be done very efficiently. When moving to the detailed level, the lower level details are taken into consideration, but the detailed diagnosis is done using the results from the previous levels. In technical systems, the hierarchical decomposition is often given through the system's physical structure. In some cases, diagnosis of the subcomponents is not even needed, if the smallest replaceable part is the aggregate component. When diagnosing configuration knowledge bases, the structural abstraction is given through the assignment of the individual constraints to component types. In other words, if we assume a component type definition to be faulty, we assume that all constraint types assigned to that component type are faulty. In the following we only consider such a two level hierarchy, although further abstractions (grouping of several related component types) may be applicable. When using this abstraction mechanism, the top level diagnoses for the problem are

$$ab(PC). \; ab(Motherboard) \wedge ab(CPU).$$

These initial diagnoses can be computed very efficiently since we have only four top level diagnosis components. Given this initial diagnosis the user can decide to focus on the *PC* component type (maybe because he/she only made changes for that component type) or can decide to iteratively refine the results through replacement of the component types with their individual constraints. In the following section we will treat this model more formally.

3 DIAGNOSIS OF CONFIGURATOR KNOWLEDGE BASES

First, we will shortly review the logical definition of configuration and configuration problems from [7]: Typically, configurable products are built from a predefined set of component types, which are characterized through attributes and can be interconnected via predefined connection points (ports). This information and the constraints on legal constellations can be expressed as a set of logical sentences *DD (domain description)*. Configuration problems have to be solved according to some specific requirements, again described through a set of sentences *SRS*. A configuration result is described by a set of ground literals containing information on the employed component instances, attribute valuations, and connections (*e.g., type, val,* and *conn* - literals).

Definition (Configuration Problem): *A configuration problem is described by a triple (DD,SRS,CONL), where DD and SRS are sets of logical sentences and CONL is a set of predicate symbols.*

DD represents the domain description, SRS an individual configuration problem instance. A configuration CONF is described by a set of ground literals whose predicate symbols are in CONL. ❑

Definition (Consistent Configuration): *Given a configuration problem (DD,SRS,CONL), a configuration CONF is consistent iff* $DD \cup SRS \cup CONF$ *is satisfiable.* ❑

To ensure the completeness of a configuration, additional formulae for each predicate symbol in *CONL* have to be introduced to *CONF*, e.g.,

$$type(X,Y) \Rightarrow type(X,Y) \in CONF.$$

We denote the configuration *CONF* extended by these axioms with $\overline{CONF}$. (For a detailed exposition, see [7]).

Definition (Valid and irreducible configuration): *Let (DD,SRS,CONL) be a configuration problem. A configuration CONF is valid iff* $DD \cup SRS \cup \overline{CONF}$ *is satisfiable. CONF is irreducible if there exists no other valid configuration* $CONF^{sub}$ *such that* $CONF^{sub} \subset CONF$. ❑

In MBD terms, a system consists of a set of components and a set of observations describing the actual behavior of the system. The role of the components is played by the elements of *DD*, the observations are given in terms of positive and negative examples.

Definition (CKB-Diagnosis Problem): *A CKB-Diagnosis Problem (Configuration Knowledge Base) is a triple (DD,*E^+, E^-*) where DD is a configuration knowledge base,* E^+ *is a set of positive and* E^- *a set of negative examples. The examples are given as sets of logical sentences. We assume each example on its own to be consistent.* ❑

Positive examples are (partial) configurations, which should be accepted by the configurator, whereas negative examples should be rejected.

Given these example sets and the domain description cause an inconsistency, a diagnosis corresponds to the removal of possibly faulty sentences restoring the consistency. In addition, if a negative example is consistent with the knowledge base, we have to find an extension to *DD* which restores inconsistency for all such negative examples.

Definition (CKB-Diagnosis): *A CKB-Diagnosis for a CKB-Diagnosis Prolem (DD,E+,*E^-*) is a set* $S \subseteq DD$ *such that there exists an extension EX, where EX is a set of logical sentences, such that*

$$DD - S \cup EX \cup e^+ \text{ consistent } \forall e^+ \in E^+$$
$$DD - S \cup EX \cup e^- \text{ in consistent } \forall e^- \in E^- \text{ ❑}$$

Proposition: *Given a CKB-Diagnosis Problem (DD,* E^+, E^-*) , a diagnosis S exists iff*

$$\forall e^+ \in E^+: \cup \bigwedge_{e^- \in E^-} (\neg e^-) \text{ is consistent.}$$

From here on we refer to the conjuction of the negated negative examples as *NE, i.e., NE =* $\bigwedge_{e^- \in E^-} (\neg e^-)$.

Proof. see [7].

In order to precisely define our approach of using abstraction hierarchies for the diagnosis task, we employ the concept of *Theory Diagnosis* from [9]. In our example, the abstraction of individual constraints from the knowledge base to component types comprising all assigned constraints can be expressed in terms of equivalences, e.g.,

$$ab(MB) \equiv ab(MB_1) \vee ab(MB_2).$$
$$ab(CPU) \equiv ab(CPU_1) \vee \ldots \vee ab(CPU_n).$$

This equivalences express that a component type definition is considered faulty, if at least one of the assigned constraints is faulty. We call this theory of hierarchical abstraction an *ab*-theory Σ.

Definition (Hierarchical Theory): *A hierarchical theory is a set of ab-clauses of the form*

$$ab(c_i) \equiv ab(c_j) \vee \ldots. \vee ab(c_k).$$

where all literals contained in such an identity are unique. There are no two equivalences having the same literal on the left hand side. ❑

Note, that such a hierarchical theory defines a set of directed trees. Although in our example, only two level hierarchies are defined, further abstractions are possible, e.g., grouping of several component type definitions to a package, i.e., a more complex substructure of the configured system. For the hierarchical diagnosis task, the domain description *DD* is extended with this abstraction hierarchy Σ.

Let $LHS(\Sigma)$ be the set of literals that appear on the left hand side of a clause from Σ. In the following, given a set $C \subseteq LHS(\Sigma)$, we denote *succ(C)* as the union of the set of all direct and indirect successors and *leaves(C)* be the union of all leaf nodes of all $c_i \in C$ in the tree formed by the hierarchical theory. In the following, *leaf constraints* denote the set of all leaves of all trees and *abstract constraints* be the non-leaf sentences.

For hierarchical diagnosis we extend our notion of *CKB-Diagnosis* in a way that also constraints can appear in the diagnosis which are on the left hand side of a clause from the hierarchical theory.

Definition (Abstract CKB-Diagnosis): *An Abstract CKB-Diagnosis for a configuration problem* (DD,E^+,E^-) *and a hierarchical theory* Σ *is a set of sentences S from DD and a non-empty set C from LHS(*Σ*) such that:*

$DD - S - leaves(C) \cup EX \cup e^+$ *consistent* $\forall e^+ \in E^+$,
$DD - S - leaves(C) \cup EX \cup e^-$ *inconsistent* $\forall e^- \in E^-$ *where*
$S \cap leaves(C) = \emptyset$, $\forall c \in C: c \notin succ(C)$. ❑

Following this definition, such a diagnosis can therefor contain individual constraints from *DD* as well as *abstract* constraints. If an abstract constraint is considered faulty, all its sub-constraints are also considered faulty and must not be contained in *S*. In addition, if an abstract constraint is considered faulty, no abstract subconstraint according to the hierarchy may be contained in *C*.

Definition (Minimal Abstract CKB-Diagnosis): *An Abstract CKB-Diagnosis* $AD = \{S \cup C\}$ *for* (DD, E^+,E^-) *and* Σ *is said to be minimal, if no subset* $AD' \subset AD$ *is an Abstract CKB-Diagnosis.* ❑

Similar to *(Minimal) Theory Diagnoses* [9], these abstract diagnoses can be used as a generator for a set of minimal diagnoses. The abstraction theory Σ describes how the abstract diagnoses have to be extended in order to provide minimal diagnoses.

4 COMPUTING DIAGNOSES

4.1 Computing abstract diagnoses

Given the above definitions we can extend the standard algorithm for consistency-based diagnosis to calculate minimal (abstract) diagnoses. The basic algorithm is extended to fit the needs of our application domain and to support the notion of structural abstraction. In the standard algorithm ([13],[17]), the concept of *conflict sets* is used for focusing purposes:

Definition (Conflict Set): *A conflict set CS for* (DD, E^+,E^-) *is a set of elements from DD such that* $\exists e^+ \in E^+$: $CS \cup e^+ \cup NE$ *is inconsistent.* ❑

In order to support the calculation of minimal abstract diagnoses, we extend the definition of conflict sets to allow a conflict set to contain not only sentences from the knowledge base but also abstract constraints, which are then replaced by their subcomponents:

Definition (Abstract Conflict Set): *An abstract conflict set ACS for* (DD,E^+,E^-) *and a hierarchical theory* Σ *is a set of elements S from DD and a set C from LHS(*Σ*) such that* $\exists e^+ \in E^+$: $S \cup leaves(C) \cup e^+ \cup NE$ *is inconsistent, where* $S \cap leaves(C) = \emptyset$, $\forall c \in C: c \notin succ(C)$. ❑

In the algorithm, the basic HS-DAG algorithm from ([13],[17]) is extended as follows: a node *n* in the tree is labeled by a conflict set *CS(n)*; edges leading away are labeled by elements $s \in CS(n)$. The set of edge labels on the path from the root to a node *n* is referred to as *H(n)*. In addition, for each node *n* a set *CE(n)* of consistent positive examples is stored, having in mind that once an example is already consistent it will not become inconsistent after further removal of constraints. Since a node can have multiple direct predecessors [13] - referred to as *preds(n)* - we combine the sets *CE* from all direct precedessors for such a node.

According to the idea of iteratively substantiating abstract diagnoses following the hierarchical structure of the problem, we will initally compute a set of high level diagnosis which can then be refined to a more detailed level. Consequently, the diagnostic algorithm has an additional input parameter beside the problem description and the examples, i.e., an abstract diagnosis that was already calculated on a higher abstraction level. During the calculation of a diagnosis the results from that higher level are reused and the diagnosis task is performed on the next detailed level, i.e., abstract constraints are replaced by the constraints from the next lower level.

Accordingly, given an abstract diagnosis *AD* as input, conflict sets returned by the call to the theorem prover must obey certain restrictions. Given a set *AC* of abstract constraints, let *sons(AC)* be the set of direct successors according to the hierarchical theory Σ.

- If $AD = \emptyset$, only sentences from the top level of the abstraction hierarchy can be contained in the returned abstract conflict set.
- If $AD \neq \emptyset$, only sentences from *sons(AD)*, constraints from the top level of the abstraction hierarchy which are not part of the hierarchy of elements of *AD*, and elements from *AD* itself, which are already at the lowest level of the abstraction theory, can be contained in the returned abstract conflict set.
- The returned (abstract) conflict set cannot contain any elements from the path *H(n)* from the root node of the HS-DAG to the current node.

Algorithm: (schema)
In: (DD,E^+,E^-), Σ, an abstract Diagnosis *AD*
Out: a set of refined diagnoses *RD*

(1) Use the hitting set algorithm to generate a pruned HS-DAG *D* for the collection *F* of abstract conflict sets for $((DD,E^+,E^-), \Sigma)$. Compute the DAG in breadth-first manner in order to generate diagnoses in order of their cardinality.

(a) Every theorem prover call $TP(DD - H(n), E^+ - CE(preds(n)), E^-, \Sigma, AD)$ at a node *n* tests whether there exists an $e^+ \in E^+$ such that there is an inconsistency. In this case an (abstract) conflict set is returned, otherwise it returns ok.

(b) Set *CE(n)* to be the set of examples found to be consistent in the call to *TP* union the examples that were already consistent at the direct precedessors of *n*.

(2) Return {H(n) | n is a node of D labeled by ok

4.2 Computing all minimal diagnoses

As described in [9], the calculated abstract diagnoses can be used to describe all minimal diagnoses and serve as a generator for all these diagnoses. Following the explanations from Section 2, we propose an iterative approach where one starts with a high level diagnosis which can be computed very efficiently and decide afterwards either to stop at the current level and focus on some package of sentences from the knowledge base or to refine the diagnosis to a more concise level. In addition, the hierarchical approach can also be applied if the

user only is interested in the most detailed level, because the hierarchical approach outperforms a flat approach in many cases. The recalculation of diagnoses at the different levels causes additional costs but the resulting HS-DAG's size is much smaller depending on the structure of the knowledge base. In addition, in cases where the employed inference engine lacks sufficient explanation facilities (see experimental results) and the calculation of minimal conflict sets would be costly, the algorithm performs better than a flat approach.

The following algorithm sketch shows how the hierarchical approach can be utilized to calculate all minimal diagnosis. In the algorithm a tree of sets of diagnoses with refinement at each level is generated. For ease of presentation, we use a *list* data structure (*DiagsList*) to hold the individual nodes (diagnoses) of the tree.

Algorithm (sketch)
DiagsList = [];
D = diagnose(DD,E^+,E^-,Σ, $\emptyset$)
/ Compute a set D of initial diagnoses on the most abstract level */*
Append all d $\in$ D to DiagsList;
index = begin of DiagsList;
set $E^+ = E^+ - \{e^+ \in E^+ \mid e^+$ consistent with DD}
/ remove examples which are always consistent with DD */*
***while** not at end of DiagsList*
  set d = diagnosis of current node;
  ***if** d is expandable*
   newD = diagnose(DD,E^+,E^-, Σ,d);
   ***for all** nd $\in$ newD*
    ***if** nd is not already in list;*
     append nd to DiagsList;
    endif
   end for
  move index to next element in DiagsList;
end while

After the calculation of a diagnosis on the most abstract level, for each resulting diagnosis a node is generated and added to the list of nodes to be refined. Remove all positive examples which are definitely consistent with the domain description. A node is *expandable*, if the diagnosis of that contains sentences which are not leaf nodes of the hierarchical theory. A node is expanded by calculating a new set of diagnoses in the context of an abstract diagnosis according to the algorithm in Section 4.1. For each of the refined diagnoses a new node is generated if that diagnosis is not already somewhere in the tree (list). The algorithm ends, if no more node can be further expanded. The algorithm prunes out elements from the tree which were already computed in another part of the tree.

The following example illustrates the algorithm for our simplified example. As a first step, an initial diagnosis on the top level is performed, yielding in the minimal diagnosis *{PC}* and *{MB,CPU}* for the abstract conflict sets *{PC,CPU}* and *{MB,CPU}*. Given the user wants to proceed to the next level, these diagnosis, which are at the top of the tree of diagnoses, are refined calling *diagnose* with the additional parameter of the more abstract diagnosis. The call to *diagnose* with *(CPU, MB)* results in the detailed diagnoses *{MB_1, CPU_1}*, *{MB_2, CPU_1}*, and *{PC}*. Note, that the component *PC* was not refined and in addition is redundant, because it is already in the tree of diagnoses (*DiagList*). The *diagnose* call with *{PC}* returns *{PC1}* and *{MB, CPU}*, where the latter is also redundant. At this stage, the tree of diagnoses cannot be refined anymore, since we already reached the lowest level of detail. As a result, the lowest level diagnoses *{PC_1}*, *{MB_1,CPU_1}*, and *{MB_2, CPU_2}* are returned.

After the computation of all detailed diagnoses, the final tree *(DiagList)* of refined diagnoses is:

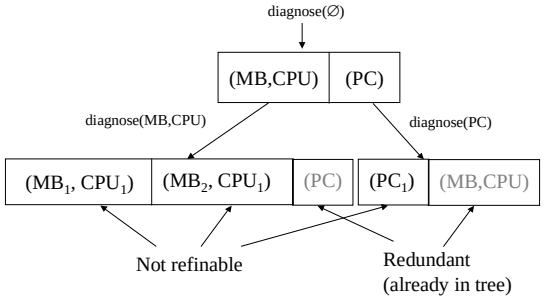


Figure 2 Tree of diagnoses for example problem

5 EXPERIMENTAL RESULTS

In order to test the applicability of our approaches, we implemented a prototype using the industrial strength software library *ILOG Configurator* [14]. Using this package of C++ libraries, a configuration problem is formulated in terms of a *Generative Constraint Satisfaction Problem* (CSP) [8]. This enhancement of the basic CSP mechanism allows the number of variables of the problem to be dynamically changed, i.e., the number of employed components may not be known beforehand. The conceptual model of this library strongly corresponds to the notion of the component port model from Section 3. Both the domain description (types, attributes, and ports), the additional constraints, and the examples can be stated using calls to the library.

In the context of that CSP, a conflict set is a set of constraints from the knowledge base, which, if canceled, makes the configuration problem satisfiable, i.e., a solution can be found. Using this library, the search for an arbitrary solution for a configuration problem can be done very efficient, especially in cases when the problem is underconstrained which is not unusual for configuration problems. However, the employed library does not offer adequate explanation mechanisms which would be helpful for the calculation of (minimal) conflict sets.

We implemented the diagnostic algorithm both for the flat approach from [7] and the extended hierarchical algorithm presented in this paper. The computation time for the diagnostis task depends on several factors such as number of constraint types, cardinality of the diagnoses or the time to test one individual example for consistency. Diagnosis of the simple example problem can be done nearly instantaneously with both algorithms; the identification of two triple faults in a setting with about twenty types of constraints and about hundred constraint instances is done in a few seconds on a standard Pentium-II PC running Windows NT with both algorithms. For these tests we employed our unoptimized prototype which does not calculate minimal conflict sets nor utilizes any special heuristics. However, for larger knowledge bases (containing about hundred types of constraints and component types), the performance of the diagnostic task using the flat approach degrades strongly when calculating diagnoses of higher cardinality. In these cases, the usage of the

hierarchical (interactive) approach with refinement of the diagnoses leverages this problem, if we assume that the given constraints are (uniformly) distributed among the component types. When considering our simple example, the *Floppy* component will only be considered as one single element of the conflict sets and will never be expanded to a more detailed level. With this approach, even larger and more complex knowledge bases remain diagnosable within an acceptable computation time, because additional constraints within the correct parts of the knowledge base only influence the costs of the consistency checks but not those of the diagnostic algorithm. In addition, we conducted experiments with real-world examples from the domain of private telecommunication systems. The tests showed that the results from the diagnostic process are suitable for debugging and validation purposes.

6 RELATED WORK

Model based diagnosis techniques were initially developed for the identification of faults in physical devices, e.g., electronic circuits. Later, these techniques were adopted for diagnosis and debugging of software, e.g., logic programs [5] or relational database consistency constraints [12]. Currently, work is underway to employ MBD techniques to debug other types of software with non-declarative semantics (e.g., hardware designs specified in VHDL [10]). [11] use model based diagnosis to find solutions for overconstrained Constraint Satisfaction Problems, which can also be achieved using our approach through the assignment of weights of importance to the individual constraints. The use of MBD techniques to solve overconstrained CSP's has also been proposed in [1].

The usage of hierarchies for the diagnosis task has been discussed in various application areas of model based diagnosis [16]. Our approach mostly corresponds to what is called *structural abstraction* (vs. *behavioral abstraction*). One of the important problems is to have the information on the hierarchy available at each abstraction level (causing additional modeling effort). For the case of debugging of constraint knowledge bases, however, the hierarchical abstraction has a good correspondence to the configurable artifact, whereas in other domains of (software) debugging, this abstraction may be more complicated to define. Furthermore, we found our logical model of configuration and our configuration language to be expressive enough to cover a wide range of configuration problems. In addition, in [6] framework is defined, where configuration knowledge bases of this type can be automatically generated from high-level conceptual product models.

7 CONCLUSIONS

The demand for AI-based product configuration technology is steadily increasing, due to the growing relevance of mass-customized production and the increasing complexity of the resulting knowledge bases. For the validation and maintenance tasks, only limited support can be found in nowadays systems. For these tasks, we show, how techniques from model based diagnosis can be utilized in supporting the knowledge engineer in validating the knowledge base. Given positive and negative examples (maybe from former configuration runs), the knowledge base is diagnosed and possible explanations for the unexpected behavior are generated. These results can serve as a pointer for the knowledge engineer, which of the constraints may have to be revised to obtain a correct knowledge base. For large and complex knowledge bases, the usage of hierarchical structural abstraction which is given by the problem structure can improve the efficiency of these diagnostic algorithms. We have sketched a basic algorithm which allows for iterative (and interactive) refinement of diagnoses, where further improvements, e.g., through reuse of conflict sets, are part of our future work.

The usage of an industrial strength configurator library alleviates the further integration of AI technology in standard software environments.

8 REFERENCES

[1] R.R. Bakker and F. Dikker and F. Tempelman and P.M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. Proc: *IJCAI'93*, p. 276–281, Chambery, Morgan Kaufmann, 1993.

[2] Virginia E. Barker and Dennis E. O'Connor. Expert systems for configuration at Digital: XCON and beyond. Comm. ACM, 32 (3): 298-318, 1989

[3] G. W. Bond. Top-down consistency based diagnosis. In Proceedings *DX'96 Workshop*, Val Morin, Canada, 1996

[4] D. Brady, K. Kerwin, D. Welch et al.: Customizing for the masses, Business Week, No. 3673, March 2000.

[5] L. Console, G. Friedrich, and D.T. Dupré: Model-based diagnosis meets error diagnosis in logic programs. In Proc. *IJCAI'93*, Chambery, Morgan Kaufmann, 1993.

[6] A. Felfernig, G. Friedrich, and D. Jannach:, UML as domain specific language for the construction of knowledge based configuration systems, in Proceedings: SEKE'99, 1999.

[7] A. Felfernig, G.Friedrich, D. Jannach, and M. Stumptner, Consistency based diagnosis of configuration knowledge-bases. In Proceedings: ECAI'2000, Berlin, August, 2000.

[8] G. Fleischanderl, G. Friedrich, A. Haselboeck, H. Schreiner and M. Stumptner, Configuring Large Systems Using Generative Constraint Satisfaction, IEEE Intelligent Systems, July/August, 1998.

[9] G. Friedrich, Theory Diagnosis: A Concise Characterization of Faulty Systems, in: Proc. IJCAI'93, Chambery, France,1993.

[10] G. Friedrich, M. Stumptner, and F. Wotawa: Model-Based Diagnosis of Hardware Designs, in Artificial Intelligence, Vol. 111, Num. 2, 1999.

[11] E. Freuder, R. J. Wallace: Partial Constraint Satisfaction, Artificial Intelligence (58), 1992.

[12] M. Gertz, U. Lipeck. A Diagnostic Approach to Repairing Constraint Violations in Databases. In Proceedings *DX'95 Workshop*, Goslar, 1995.

[13] R. Greiner, B.A. Smith, and R.W. Wilkerson: A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1), 1989.

[14] D. Mailharro: A Classification and Constraint-based Framework for Configuration, AI EDAM, Vol 12 (1998), Cambridge University Press, 1998.

[15] S. Mittal and F. Frayman, Towards a generic model of configuration tasks, in Proceedings: IJCAI'89, 1989.

[16] I. Mozetic: Hierarchical Model Based Diagnosis, in: Harmscher et al.: Readings in Model Based Diagnosis, Morgan Kaufmann, 1992.

[17] R. Reiter: A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1), 1987.

[18] M. Stumptner, An overview of knowledge-based configuration, AI Communications 10(2), pages 111-126, 1997.

Integration of Distributed Constraint-Based Configurators

Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach and **Markus Zanker** [1]

Abstract. Configuration problems are a thriving application area for declarative knowledge representation that experiences a constant increase in size and complexity of knowledge bases. However, today's configurators are designed for solving local configuration problems not providing any distributed configuration problem solving functionality. Consequently the challenges for the construction of configuration systems are the integrated support of configuration knowledge base development and maintenance and the integration of methods that enable distributed configuration problem solving.

In this paper we show how to employ a standard design language (Unified Modeling Language - UML) for the construction of configuration knowledge bases (component structure and functional architecture) and automatically translate the resulting models into an executable logic representation which can further be exploited for calculating distributed configurations. Functional architectures are shared among cooperating configuration systems serving as basis for the exchange of requirements between those systems. An example for configuring cars shows the whole process from the design of the configuration model to distributed configuration problem solving.

1 Introduction

Knowledge-based configuration systems have a long history as a successful AI application area and today form the foundation for a thriving industry (e.g. telecommunication systems, automotive industry, computer systems etc.). These systems have likewise progressed from their successful rule-based origins [1] to the use of higher level representations such as various forms of constraint satisfaction [19], description logics [12], or functional reasoning [18], due to the significant advantages offered: more concise representation, higher maintainability, and more flexible reasoning. Furthermore, the increasing demand for applications providing solutions for configuration tasks is boosted by the mass customization paradigm and e-business applications.

Especially the integration of configurators in order to support cooperative configuration such as supply chain integration of customizable products is an open research issue. Current configurator approaches [7] are designed for solving local configuration problems, but there is still no support for cooperative solving of distributed configuration tasks. Security and privacy concerns as well as the impossibility to exchange constraints represented in different representation formalisms make it impossible to centralize problem solving in one configurator.

In order to meet these challenges, we propose a framework for designing and integrating configuration systems based on the interchange of functional architectures. Functional architectures [13] determine *what* can be realized by a product, i.e. specify the functions a product provides including constraints between those functions and a mapping from functions to components the final product can be built of (*how* can the functions be implemented). Mostly, customers are not interested in the detailed product topology but rather specify a set of functions the product must provide. The configurator either configures the corresponding product or informs the customer about incompatible requirements. In the following we discuss a similar scenario, in which configurators act as customers and suppliers.

The development process for configuration systems is sketched in Figure 1. The starting point is the design of the configuration knowledge base consisting of functional architectures [13] and a corresponding component structure. In order to simplify the construction of a constraint-based description of the domain knowledge we employ UML - Unified Modeling Language [17], which is a standard design language widely applied in industrial software development processes. For sharing configuration knowledge between different systems we propose the exchange of functional architectures, i.e. if a configurator wants to order products from another configurator it must integrate the functional architecture of the desired product into its local knowledge base (phase 1). The resulting conceptual configuration model is automatically translated into a representation executable by the corresponding configuration system. Since our goal is to support cooperative configuration, the translation process must generate a representation applicable by a distributed problem solving algorithm (phase 2). For guiding the problem solving process of distributed configuration we employ *asynchronous backtracking* proposed by [21], which offers the basis for bounded learning strategies supporting the reduction of search efforts (phase 3).

The paper is organized as follows. First, we briefly sketch the design of a configuration model using UML (Section 2). In Section 3 we give a formal definition of a distributed configuration task based on the component port model [13] and show how to translate a configuration model designed in UML into this formalism. In order to show the applicability of asynchronous backtracking for distributed configuration problem solving we translate the component port representation into a distributed CSP representation which can be exploited by asynchronous backtracking. In Section 4 we show how to share functional architectures between configuration systems and how to organize the local configuration knowledge in order to assure to be executable by asynchronous backtracking. Furthermore we give an example for a distributed car configuration which is realized by three configuration systems (car manufacturer, electric equipment supplier, and motor-unit supplier). Sections 5 and 6 contain related work and general conclusions.

[1] Institut für Wirtschaftsinformatik und Anwendungssysteme, Produktionsinformatik, Universitätsstrasse 65-67, A-9020 Klagenfurt, Austria, email: {felfernig,friedrich,jannach,zanker}@ifi.uni-klu.ac.at.

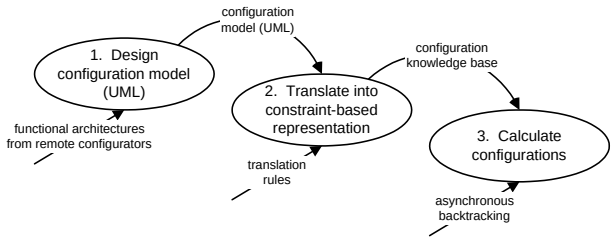


Figure 1. Configuration system development process

2 Designing configuration knowledge bases

In the following we give an overview of the modeling concepts used for designing highly variant products. For presentation purposes we introduce simplified models of a car manufacturer (Figure 2), a motor-unit supplier (Figure 3), and an electric equipment supplier (Figure 4) as a working example.

We employ the extension mechanism of UML (stereotypes) to express domain-specific modeling concepts. The semantics of the different modeling concepts are formally defined by the mapping of the graphical notation to logical sentences based on the component port model[2] (see Section 3). The basic structure of the product is modeled using classes, generalization, and aggregation of component types and function types. The following concepts are employed for designing configuration models.

- **Component types** These represent parts the final product can be built of. Component types are characterized by attributes.
- **Function types** They are used to model the functional architecture of an artifact, which can be integrated into configuration models of other configurators. Similar to component types they can be characterized by attributes.
- **Resources** Parts of a configuration problem can be seen as a resource balancing task, where some of the component types *produce* some resource and others are *consumers*.
- **Generalization** Component (function) types with a similar structure are arranged in a generalization hierarchy.
- **Aggregation** Aggregations between components (functions) represented by part-of structures state a range of how many subparts (subfunctions) an aggregate can consist of.
- **Connections and ports** In addition to the amount and types of the different components also the product topology may be of interest in a final configuration, i.e. how the components are interconnected to each other.
- **Compatibility and requirements relations** Some types of components (functions) cannot be used in the same final configuration - they are *incompatible*. In other cases, the existence of one component (function) type *requires* the existence of another special type in the configuration.
- **Functional architectures** Functional architectures represent exactly those elements of the configuration model, which can be shared between cooperating configurators. The mapping from functions to components is modeled using the *requires* relations in the simple case. More complex relationships between functions and components can either be represented by additionally defined modeling concepts or OCL (Object Constraint Language) constraints. The mapping between functions and components is *many-to-many* [13], e.g. the *lights-function* in Figure 5 is implemented by the components *front-fog-lights* and *large-battery*.
- **Additional modeling concepts and constraints** Constraints on the product model, which can not be expressed graphically, are formulated using the language OCL, which is an integral part of UML. As it is done for the graphical modeling concepts, OCL expressions are translated into a logical representation executable by the configuration engine. The discussed modeling concepts have shown to cover a wide range of application areas for configuration [16]. Despite this, some application areas may have a need for special modeling concepts not covered so far. To introduce a new modeling concept a new stereotype has to be defined. Its semantics for the configuration domain must be defined by stating the facts and constraints induced to the logic theory when using the concept.

[2] A detailed discussion on the translation rules can be found in [8].

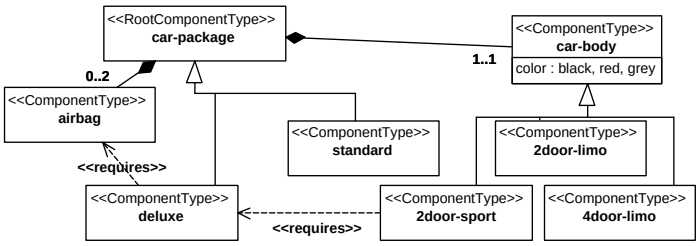


Figure 2. Component structure of car configurator

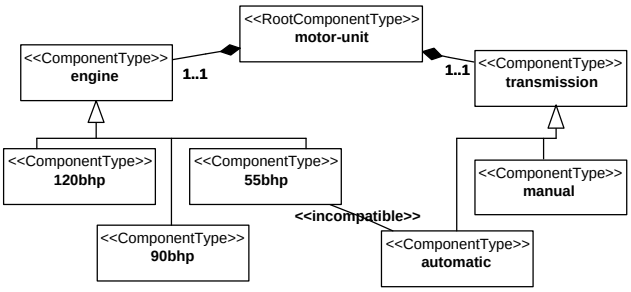


Figure 3. Component structure of motor configurator

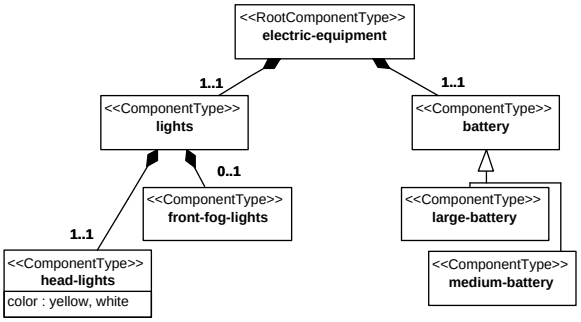


Figure 4. Component structure of electric equipment configurator

3 Distributed Configuration Task

In this section we give a formal definition of a distributed configuration task based on the component port model [13], which allows an intuitive definition using configuration domain specific representation concepts. In Section 4 we employ a constraint-based representation in order to show the distribution of constraint variables representing functional architectures.

In practice, configurations are built from a predefined catalog of component types (*types*) of a given application domain. Furthermore, the configuration task is characterized by

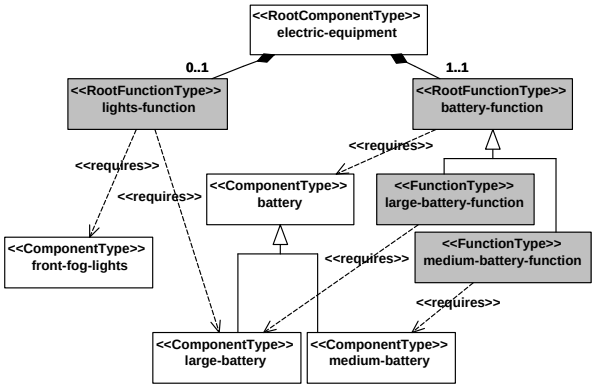


Figure 5. Functional architectures and component mapping of electric equipment supplier

a set of functional architectures which specify the functional composition of artifacts and constraints on their composition, i.e. a set of necessary and optional functions, and constraints on their composition [9], [13]. The set of functions is further denoted as *functions*. Component types as well as function types are described through a set of properties (*attributes*), and connection points (*ports*) representing logical or physical connections to other components. Both, *attributes* and *ports* have an assigned domain (*dom*).

The domain description (*DD*) of a configuration task contains this information (*types, functions, ports, attributes, dom*) and additional constraints on legal configurations. The actual configuration problem has to be solved according to the set *SRS* (system requirements specification).

The *DD* is derived by translating the component structure as well as the functional architecture(s) and the corresponding constraints. Based on this characterization of a local configuration task [9], [13] we define a *Distributed Configuration Task* through the following sets of logical sentences.

- $DD = \bigcup DD_i$, where DD_i is the *DD* of configurator i ($i \in \{1..n\}$ and n is the number of cooperating configurators).
- $SRS = \bigcup SRS_i$.

A configuration result is described through sets of logical sentences *(FUNCS, COMPS, ATTRS, CONNS)*. In these sets the employed functions, components, attribute values, and established connections of a concrete customized product are represented.

- $FUNCS = \bigcup FUNCS_i$, where $FUNCS_i$ represents sets of literals of the form *func(c,t)*. *t* is included in the set of *functions* defined in DD_i. The constant *c* represents the identifier of a function.
- $COMPS = \bigcup COMPS_i$, where $COMPS_i$ represents sets of literals of the form *type(c,t)*. *t* is included in the set of *types* defined in DD_i. The constant *c* represents the identifier of a component.
- $CONNS = \bigcup CONNS_i$, where $CONNS_i$ represents sets of literals of the form *conn(c1,p1,c2,p2)*. *c1, c2* are component (function) identifiers from $COMPS_i$ (FUNCS_i). *p1 (p2)* is a port of the component (function) *c1 (c2)*.
- $ATTRS = \bigcup ATTRS_i$, where $ATTRS_i$ represents sets of literals of the form *val(c,a,v)*, where *c* is a component (function) identifier, *a* is an attribute of that component (function), and *v* is the actual value of the attribute.

The *DD* of the electric equipment supplier (Figure 4 and Figure 5) is the following:

types={electric-equipment,lights,battery,front-fog-lights,
head-lights,large-battery,medium-battery}.
functions={lights-function, battery-function,
medium-battery-function, large-battery-function}.
attributes(head-lights)={color}.
dom(head-lights, color)={yellow, white}.
ports(electric-equipment)={lights-port, battery-port,
lights-function-port,battery-function-port}[3].
dom(electric-equipment, lights-port)={electric-equipment-port}
ports(lights)={electric-equipment-port}.
ports(battery)={electric-equipment-port}.
ports(lights-function)={electric-equipment-port}.
ports(battery-function)={electric-equipment-port}. ...

The relation *lights-function requires large-battery* (Figure 5) is translated as follows[4]:

type(ID1, lights-function) ∧
conn (ID1, electric-equipment-port, ID2, lights-function-port) ⇒
∃(ID3) type(ID3, large-battery) ∧
conn (ID3,electric-equipment-port, ID2, battery-port).

The relation *55bhp incompatible automatic* in Figure 3 is translated as follows:

type(ID1, 55bhp) ∧ conn (ID1, motor-unit-port, ID2, engine-port) ∧
conn(ID2, transmission-port, ID3, motor-unit-port) ∧
type(ID3, automatic) ⇒ false.

An example for a configuration result of the electric equipment supplier is the following:

type(electric-equipment-1, electric-equipment). type(lights-1, lights).
type(head-lights-1, head-lights). type(battery-1, medium-battery).
func(battery-function-1, medium-battery-function).
conn(head-lights-1, lights-port, light-1, head-lights-port).
conn(lights-1, electric-equipment-port, electric-equipment-1, lights-port).
conn(battery-1, electric-equipment-port, electric-equipment-1, battery-port).
conn(battery-function-1, electric-equipment-port, electric-equipment-1,
battery-function-port).

The concept of a *Consistent Distributed Configuration* is defined as follows:

Definition 1: Consistent Distributed Configuration. *If (DD, SRS) is a configuration problem and FUNCS, COMPS, CONNS, and ATTRS represent a configuration result, then the configuration is consistent exactly iff* $DD \cup SRS \cup FUNCS \cup COMPS \cup CONNS \cup ATTRS$ *can be satisfied.*

We specify that *FUNCS* includes all required functions, *COMPS* includes all required components, *CONNS* describes all required connections, and *ATTRS* includes a complete value assignment to all variables in order to achieve a complete distributed configuration[5]. Let AX_{comp} be the additional sentences for completeness purpose.

In order to assure completeness and correctness of the distributed configuration w.r.t. the overall configuration task the following sentence must hold:

- $DD \cup SRS \cup FUNCS \cup COMPS \cup CONNS \cup ATTRS \cup AX_{comp}$ *is consistent iff* $\forall i : DD_i \cup SRS_i \cup FUNCS \cup COMPS \cup CONNS \cup ATTRS \cup AXcomp$ *is consistent.*

This sentence is fulfilled if we allow in *DD* only sentences using *func, type, conn*, and *val* literals since $FUNCS \cup COMPS \cup CONNS \cup ATTRS \cup AX_{comp}$ is a complete theory w.r.t.

[3] The *part of* relationships between component and function types are translated into connections between component/function ports in the component port representation.

[4] The form of the sentences is restricted to a subset of range-restricted first-order-logic with set extension and interpreted function symbols. The term-depth is restricted to a fixed number in order to assure decideability. Additionally domain specific axioms are added, e.g. one port can only be connected to exactly one other port.

[5] This is accomplished by additional logical sentences which can be generated using the domain description (see [9] for more details).

these literals. A distributed configuration, which is consistent and complete w.r.t. the domain description and the customer requirements, is called a *Valid Distributed Configuration*.

In order to calculate solutions for a given distributed configuration task we employ *asynchronous backtracking* [21], which offers the basis for bounded learning strategies supporting the reduction of search efforts. This efficient revision of requirements and design decisions is of particular interest for integrating configurators, since supplier configurators eventually discover conflicting requirements (*nogoods*) which must be communicated back to the requesting configurator.

4 Variable Partitioning for Asynchronous Backtracking

4.1 Distributing Functional Architectures

In order to enable effective distributed configuration, configuration knowledge must be shared between configurators. In the following we show how knowledge sharing can be realized by exchanging functional architectures.

Definition 2: Functional Architecture. Let FA_{ij} be the *<<RootFunctionType>> j* of configuration model *i*, which is a direct part of the *<<RootComponentType>>* of the configuration model *i*, then FA_{ij} is a *functional architecture*, which includes the *function types* which are directly or transitivly connected with FA_{ij} via generalizations or aggregations, the corresponding *attributes*, and connected *part-of relations*. Furthermore, all constraints exclusively concerning functions and attributes of FA_{ij}, belong to FA_{ij}.

For example, *<<RootFunctionType>> battery-function* represents a functional architecture which is a direct part of *<<RootComponentType>> electric-equipment* (see Figure 5).

Figure 6 shows the configuration model of the motor-unit supplier configurator including an integrated *battery-function* architecture imported from the electric equipment supplier, i.e. the electric equipment supplier transfers the battery configuration task to the motor-unit supplier by providing the configuration information through the functional architecture of the battery.

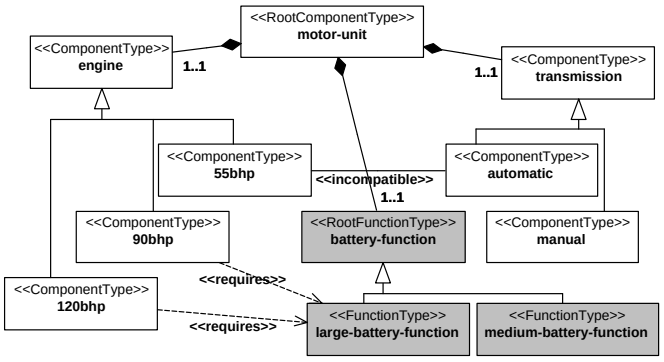


Figure 6. Imported functional architecture of motor-unit configurator

Furthermore, the electric equipment supplier exports the functional architecture *lights-function* to the car manufacturer, i.e. the car configurator is responsible for communicating requirements concerning lights to the electric equipment supplier. Finally, the functional architecture for configuring a *motor-unit* (*motor-unit-function*) is exported to the car-manufacturer. Figure 7 gives an overview of the distribution of functional architectures between the car manufacturer, electric equipment supplier, and the motor-unit supplier[6].

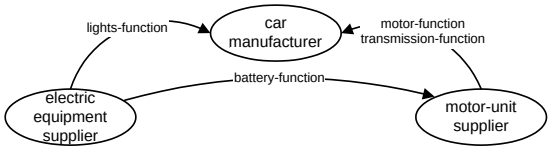


Figure 7. Distribution of functional architectures

4.2 Asynchronous Backtracking for Distributed Configuration

In this section we employ a constraint-based representation of the electric equipment configurator knowledge base in order to show the distribution of constraint variables for asynchronous backtracking. The *[priority]* represents the global priority of a variable, e.g. *[c1]* denotes the priority value of a variable of configurator *c* with priority 1, i.e. *[priority]* is a combination of configurator priority and local priority. Furthermore, the shared configuration knowledge must be represented in a shared name space, e.g. the electric equipment configurator must interpret a variable *battery-1* the same way as the motor-unit configurator. In order to represent the shared configuration knowledge in a shared name space we employ a shared naming strategie for constraint variables. In the following example the variable names are constructed by *<function-type-name/component-type-name>+local-id*, where *local-id* distinguishes between different variables of one component/function type, e.g. *airbag-2* would represent the second instance of the component type *airbag* (see Figure 2). Inactivity states of variables are represented as *novalue* in a variable's domain. We use the following set of constraint variables in order to discuss variable partitioning for asynchronous backtracking. We do not go into further details on the translation of the component port representation into a constraint-based representation. A detailed discussion on this topic can be found in [20], [19]. The following variables represent type variables derived from the configuration model of the electric equipment supplier, e.g. the variable *battery-1* can be instantiated with *medium-battery* or *large-battery*. Furthermore *front-fog-lights* are optional in the final configuration - this choice is represented by the domain *{lights-function, novalue}* of the variable *front-fog-lights-1*. For simplicity we omit the corresponding attribute and port variables.

[c1]electric-equipment-1:{electric-equipment},
[c2]lights-1:{lights},
[c3]battery-1:{medium-battery, large-battery},
[c4]front-fog-lights-1:{front-fog-lights, novalue},
[c5]head-lights-1:{head-lights},
[c6]lights-function-1: {lights-function, novalue},
[c7]battery-function-1: {medium-battery-function,
large-battery-function}.

After having distributed the configuration knowledge by exchanging and integrating functional architectures (on a conceptual level) we must define rules for how to organize the local configuration knowledge in order to employ asynchronous backtracking for calculating a solution for a given distributed configuration task.

Asynchronous backtracking proposed by [21] is an algorithm calculating solutions for distributed constraint satisfaction problems (DCSP's), where problem variables are distributed among problem solving agents and each agent has exactly one variable. Each agent has a unique priority. Constraints are directed between the variables in the sense that one of the connected agents is the *value sending agent* (agent, which

[6] Note that the functional architectures of the motor-unit configurator and the car configurator are not part of the presented UML models.

locally changes the variable instantiation), the other one is the *constraint evaluating agent* which informs the value sending agent about local inconsistencies. Changes of local assignments are communicated to constraint evaluating agents via *ok?* messages, inconsistent variable assignments are comunicated to value sending agents via *nogood* messages. Variable assignments of value sending agents are stored in the local *agent_ view*, which is used to check the consistency of local variable instantiations with higher priority variables. *Nogoods* represent conflicting variable instantiations which are calculated by applying resolution. Value sending agents have a higher priority than connected constraint evaluating agents.

In order to employ asynchronous backtracking for solving distributed configuration tasks the following requirements must hold:

1. **Configurator:** Each configurator has a set of variables representing the functions, components, ports, and attributes of our logical notation.
2. **Ordering of variables:** If a functional architecture of configuration model i is exported to configuration model k, no functional architecture from k can be integrated in i. Regarding the corresponding configurators, *configurator i* is the supplier configurator and *configurator k* is the consumer configurator. Consequently, we can derive a total order on all variables from the existing partial ordering on the involved configurators. The partial order is a result of the non-ambiguous consumer-supplier relationship between each pair of connected configurators, where all variables of the consumer must have higher priority than those of the producer.
3. **Constraint evaluating configurators:** Let V_j be the set of variables derived from functional architecture FA_{ij} of configuration model i imported from configuration model k *($k \neq i$)*. Then each variable $V \in V_j$ is evaluated by configurator k *(constraint evaluating configurator)*, i.e. is represented in the *agent_ view* of configurator k.
4. **Value sending configurators:** Let V_j be the set of variables derived from functional architecture FA_{kj} of configuration model k exported to configuration model i *($i \neq k$)* Then each variable $V \in V_j$ is represented in the local *agent_ view* of *configurator k* through a copy of V, which is updated by the *value sending configurator i*.

Figure 8 shows a DCSP representation for the *electric-equipment* configurator knowledge base including variables derived from functions and component types (attributes and ports are omitted for simplicity).

The electric equipment configurator has a set of local variables derived from those parts of the configuration model (represented in UML), which were not exported to other configuration models (variables *electric-equipment-1, front-fog-lights-1, battery-1, head-lights-1, lights-1*). No functions are element of the local variable set, since all functional architectures were exported to other configuration models. Furthermore, the variable *lights-function-1* (car configuator) is derived from the functional architecture *lights-function* imported from the electric equipment supplier, and *battery-function-1* (motor-unit configurator) is derived from the functional architecture *battery-function* imported from the electric equipment supplier. The local *agent_ view* of the electric equipment supplier contains a copy of both variables.

4.3 Example for Distributed Configuration

In order to illustrate the concepts discussed so far, we now give an example for solving a distributed configuration task using asynchronous backtracking. In the following we focus

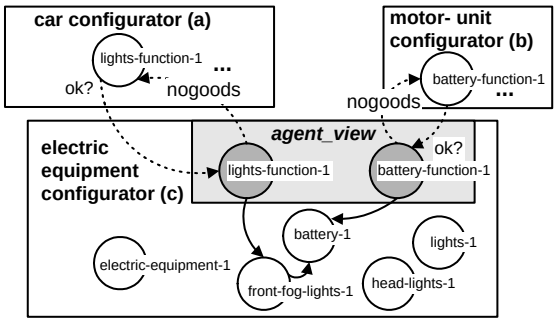


Figure 8. Interface of electric equipment configurator

on the interaction between motor-unit configurator and electric equipment configurator. Having calculated a configuration conform to the requirements of the car configurator, the motor-unit configurator communicates the following requirements to the electric equipment configurator:

ok?((battery-function-1,medium-battery-function)).

Furthermore, the electric equipment configurator receives the following requirements from the car configurator:

ok?((lights-function-1, lights-function)).

The electric equipment configurator tries to calculate a local solution and detects a contradiction between the *lights-function* and the *medium-battery-function*, since the *lights-function requires* a *large-battery* component, whereas the *medium-battery-function* requires a *medium-battery* component. Consequently a *nogood* message is sent to the motor-unit configurator:

nogood((battery-function-1,medium-battery-function),
(lights-function-1,lights-function)).

The motor-unit configurator locally stores the *nogood* and calculates an alternative solution, i.e. chooses the *large-battery-function*. The new functional requirements are communicated to the electric equipment configurator:

ok?((battery-function-1,large-battery-function)).

Finally the electric equipment configurator calculates a solution regarding the requirements of the car configurator and the motor-unit configurator.

The soundness and completeness of asynchronous backtracking is shown in [21]. The worst-case time complexity of asynchronous backtracking is exponential in the number of variables. The worst-case space complexity depends on the strategy we employ to handle *nogoods*. The options range from unrestricted learning (e.g. storing all *nogoods*) to the case where *nogood* recording is limited as much as possible. For each configurator it is sufficient to store only one *nogood* for each $d \in Dom(x_i)$ for each configurator variable x_i. These *nogoods* are needed to avoid a subsequent assignment of the same value for the same search space. In addition the generation of *nogoods* is another source of high computational costs. *Nogoods* need not be minimal, i.e. for the *nogood* generation even the complete *agent_ view* is an acceptable *nogood*. However, non-minimal *nogoods* lead to higher search efforts. The advantages and strategies for exploiting *nogoods* to limit the search activities are discussed in [2], [6].

5 Related Work

There is a broad spectrum of representation formalisms employed in knowledge-based configuration systems [1], [10], [12], [18], [19]. Todays configurators are tailored to solve local configuration tasks and there is no support for integrating these systems in order to allow cooperative configuration. Furthermore, the increasingly complex tasks tackled by configuration

systems require the provision of methods for designing configuration knowledge bases on a higher level of abstraction also understandable by technical experts.

The automated generation of logic-based descriptions through translation of domain specific modeling concepts expressed in terms of a standard design language like UML has not been discussed so far. Comparable research has been done in the fields of automated and knowledge-based software engineering [11]. In [3] a formal semantics for object model diagrams based on OMT is defined in order to support the assessment of requirement specifications. We view our work as complementary since our goal is the generation of executable logic descriptions.

An overview on aspects and applications of functional representations is given in [4], where the functional representation of a device is divided into three parts. The intended function, the structure of the device, and a description how the device achieves a function represented through a process description. [13] propose the integration of functional architectures into the configuration model by defining a matching from functions to key components, which must be part of the configuration if the function should be provided. Exactly this interpretation for the achievement of functions is used in our framework for the integration of configuration systems.

Designing large scale products requires the cooperation of a number of different experts. In the SHADE (Shared Dependency Engineering) project [15] a KIF [14] formalism was used for representing engineering ontologies. Giving an example of a spring construction, the integration of a project engineering agent responsible for the definition of the component hierarchy and basic properties of mechanic components, a spring design agent responsible for the design of the detailed technical structure and an optimization agent is shown. This approach differs from what we did in the sense that no high level design representations are provided to represent the distributed design task, furthermore no strategies for knowledge sharing between the cooperating agents are proposed.

In [5] an agent architecture for solving distributed configuration-design problems is proposed. The whole problem is decomposed into sub-problems of manageable size which are solved by agents. The primary goal of this approach is efficient distributed design problem solving, whereas our concern is to provide effective support of distributed configuration problem solving, where knowledge is distributed between different agents having a restricted view on the whole configuration process.

6 Conclusions

In order to support supply chain integration of configurable products solutions are required for integrating local configuration systems, which allow cooperative configuration problem solving. An important precondition for solving this integration task is the definition of terms and conditions representing common concepts understandable by the corresponding configuration systems. In this paper we have proposed a framework for modeling configuration knowledge bases using a standard design language and integrating the resulting knowledge bases by the exchange of functional architectures, which represent the interface between those configuration systems. Furthermore, we have shown how to translate models represented in UML in order to be executable by algorithms based on bounded learning strategies such as asynchronous backtracking. The concepts presented in this paper are an essential part of an integrated environment for the development of cooperative configuration systems, where configuration models represented in UML are automatically translated into the constraint representation of ILOG Solver.

REFERENCES

[1] V.E. Barker, D.E. O'Connor, J.D. Bachant, and E. Soloway. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, 32, 3:298–318, 1989.

[2] R.J. Bayardo and D.P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proc. AAAI*, pages 298–304, Portland, Oregon, 1996.

[3] R.H. Bourdeau and B.H.C. Cheng. A formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21,10:799–821, 1995.

[4] B. Chandrasekaran, A. Goel, and Y. Iwasaki. Functional Representation as Design Rationale. *IEEE Computer, Special Issue on Concurrent Engineering*, pages 48–56, 1993.

[5] T.P. Darr and W.P. Birmingham. An Attribute-Space Representation and Algorithm for Concurrent Engineering. *AIEDAM*, 10,1:21–35, 1996.

[6] R. Dechter. Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence*, 40,3:273–312, 1990.

[7] B. Faltings, E. Freuder, and G. Friedrich, editors. Workshop on Configuration. *AAAI Technical Report WS-99-05*, Orlando, Florida, 1999.

[8] A. Felfernig, G. Friedrich, and D. Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. In *11th International Conference on Software Engineering and Knowledge Engineering*, pages 337–345, Kaiserslautern, Germany, 1999.

[9] G. Friedrich and M. Stumptner. Consistency-Based Configuration. In *AAAI Workshop on Configuration, Technical Report WS-99-05*, pages 35–40, Orlando, Florida, 1999.

[10] M. Heinrich and E.W. Jüngst. A resource-based paradigm for the configuring of technical systems from modular components. In *Proc. 7th IEEE Conference on AI applciations (CAIA)*, pages 257–264, Miami, FL, USA, 1991.

[11] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A Formal Approach to Domain-Oriented Software Design Environments. In *Proceedings 9th Knowledge-Based Software Engineering Conference*, pages 48–57, Montery, CA, USA, 1994.

[12] D.L. McGuiness and J.R. Wright. Conceptual Modeling for Configuration: A Description Logic-based Approach. *AIEDAM, Special Issue: Configuration Design*, 12,4:333–344, 1998.

[13] S. Mittal and F. Frayman. Towards a Generic Model of Configuration Tasks. In *Proc. of the 11th IJCAI*, pages 1395–1401, Detroit, MI, 1989.

[14] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12,3:36–56, 1991.

[15] G.R. Olsen, M. Cutkosky, J.M. Tenenbaum, and T.R. Gruber. Collaborative Engineering based on Knowledge Sharing Agreements. In *Proceedings of ACME Database Symposium*, pages 11–14, Minneapolis, MN, USA, 1994.

[16] H. Peltonen, T. Männistö, T. Soininen, J. Tiihonen, A. Martio, and R. Sulonen. Concepts for Modeling Configurable Products. In *Proceedings of European Conference Product Data Technology Days*, pages 189–196, Sandhurst, UK, 1998.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[18] J.T. Runkel, A. Balkany, and W.P. Birmingham. Generating non-brittle configuration-design tools. *Artificial Intelligence in Design, Kluwer Academic Publisher*, pages 183–200, 1994.

[19] M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2), June, 1997.

[20] M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *AIEDAM, Special Issue: Configuration Design*, 12, 4:307–320, Sep. 1998.

[21] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem. *IEEE Transactions on Knowledge and Data Engineering*, 10,5:673–685, 1998.

Distributed Configuration as Distributed Dynamic Constraint Satisfaction

Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach and **Markus Zanker** [1]

Dynamic constraint satisfaction problem (DCSP) solving is one of the most important methods for solving various kinds of synthesis tasks, such as configuration. Todays configurators are standalone systems not supporting distributed configuration problem solving functionality. However, supply chain integration of configurable products requires the integration of configuration systems of different manufacturers, which jointly offer product solutions to their customers. As a consequence, we need problem solving methods that enable the computation of such configurations by several distributed configuration agents. Therefore, one possibility is the extension of the configuration problem from a dynamic constraint satisfaction representation to *distributed* dynamic constraint satisfaction (DDCSP). In this paper we will contribute to this challenge by formalizing the DDCSP and by presenting a complete and sound algorithm for solving distributed dynamic constraint satisfaction problems. This algorithm is based on asynchronous backtracking and enables strategies for exploiting conflicting requirements and design assumptions (i.e. learning additional constraints during search). The exploitation of these additional constraints is of particular interest for configuration because the generation and the exchange of conflicting design assumptions based on *nogoods* can be easily integrated in existing configuration systems.

1 Introduction

Dynamic constraint satisfaction is a representation formalism suitable for representing and solving synthesis tasks, such as configuration [11], [13]. These tasks have a dynamic nature in the sense that the set of problem variables changes depending on the initial requirements and the decisions taken during the problem solving process resulting in a reduction of the search space to relevant problem variables. In this context the notion of activity constraints discussed in [11] is well suited for stating decision criteria on the activity state of problem variables.

There is an increasing demand for applications providing solutions for configuration tasks in various domains (e.g. telecommunications industry, computer industry, or automotive industry). This demand is boosted by the mass customization paradigm and e-business applications. Especially the integration of configurators in order to support cooperative configuration such as supply chain integration of customizable products is an open research issue. Current configurator approaches [3] , [7] are designed for solving local configuration problems, but there is still no problem solving method which considers variable activity state in the distributed case. Security and privacy concerns make it impossible to centralize problem solving in one centralized configurator.

As a consequence, we have to extend dynamic constraint satisfaction to *distributed* dynamic constraint satisfaction. Based on asynchronous backtracking [14] we propose an extension for standard distributed CSP formalisms by incorporating activity constraints in order to reason about the activity states of problem variables. In the sense of distributed constraint satisfaction we characterize a Distributed Dynamic Constraint Satisfaction Problem (DDCSP) as a cooperative distributed problem solving task. Activity constraints, compatibility constraints as well as variables are distributed among the problem solving agents. The goal is to find a variable instantiation which fulfills all local as well as inter-agent constraints. In this context each agent has a restricted view on the knowledge of another agent, but does not have complete access to the remote agents knowledge base.

Asynchronous backtracking offers the basis for bounded learning strategies which supports the reduction of search efforts. This is of particular interest for integrating configurators. Configurators send configuration requests to their solution providing partners. These partners eventually discover conflicting requirements (i.e. *nogoods*) which are communicated back to the requesting configurator thus supporting the efficient revision of requirements or design decisions. Based on asynchronous backtracking we provide a sound and complete algorithm for distributed dynamic constraint satisfaction where a very limited *nogood* recording is sufficient.

In the following we give a formal definition for a DDCSP *(Section 2)* and show how to employ this formalism for representing a distributed configuration problem. Following this formal definition we propose an algorithm for solving DDCSP *(Section 3)*. We analyze runtime and space complexity of this algorithm and show completeness and soundness. Finally we discuss related work followed by general conclusions.

2 Distributed Dynamic Constraint Satisfaction Task

Before giving a definition of a DDCSP we recall the definition of a distributed CSP. A set of agents must find a solution for a distributed set of finite domain variables. There exist n agents,

[1] Institut für Wirtschaftsinformatik und Anwendungssysteme, Produktionsinformatik, Universitätsstrasse 65-67, A-9020 Klagenfurt, Austria, email: {felfernig,friedrich,jannach,zanker}@ifi.uni-klu.ac.at.

where each agent a_i has m variables x_{ij}, $i \in \{1..n\}$, $j \in \{1..m\}$. Agent variables have a priority denoted as p_{ij}. Each variable x_{ij} is assigned to a domain $D_l \in Doms = \{D_1, D_2, ...\}$, where $Dom(x_{ij})$ denotes the domain of the variable x_{ij}. Constraints are distributed among agents, where C_{ik} denotes constraint k of agent a_i. A distributed CSP is solved *iff* $\forall x_{ij}$, $\forall C_{ik}$: x_{ij} is instantiated with one $d \in Dom(x_{ij})$, and C_{ik} is true under the assignment $x_{ij}=d$.

In order to formulate constraints on the activity state of problem variables, [11] propose four major types of activity constraints as follows. Note, that $x_{ij} \notin \{x_{pq} \wedge x_{vw} \wedge \ldots\}$ must hold, where x_{ij}, x_{pq}, and x_{vw} are different agent variables.

1. *Require Variable* ($\overset{RV}{\Rightarrow}$): the activity state of a variable depends on the value assignment to a set of active variables, i.e. $P(x_{pq}, x_{vw}, ...) \overset{RV}{\Rightarrow} x_{ij}$. P denotes a predicate determining whether the variable x_{ij} must be active or not.
2. *Always Require* ($\overset{ARV}{\Rightarrow}$): the activity state of a variable depends on the activity state of a set of other variables, i.e. $(x_{pq} \wedge x_{vw} \wedge ...) \overset{ARV}{\Rightarrow} x_{ij}$.
3. *Require Not* ($\overset{RN}{\Rightarrow}$): a variable must not be active if a certain assignment of a set of variables is given, i.e. $P(x_{pq}, x_{vw}, ...) \overset{RN}{\Rightarrow} x_{ij}$.
4. *Always Require Not* ($\overset{ARN}{\Rightarrow}$): a variable must not be active if a set of variables is active, i.e. $(x_{pq} \wedge x_{vw} \wedge ...) \overset{ARN}{\Rightarrow} x_{ij}$.

Based on the above definition of a distributed CSP and the notion of activity constraints we give a formal definition of a DDCSP. In order to determine whether a variable x_{ij} is active or not we associate a state variable with each x_{ij} denoted as $x_{ijstatus}$, where $Dom(x_{ijstatus}) = \{active, inactive\}$. Additionally there are two different types of constraints, namely *compatibility constraints* (C_C), which restrict the compatibility of variable assignments, and *activity constraints* (C_A), which constrain the activity state of constraint variables.

Definition 1 *Distributed Dynamic CSP (DDCSP)*
Given:

- *n agents, where agent* $a_i \in \{a_1, a_2, ..., a_n\}$.
- *Each agent* a_i *knows a set of variables* $V_i = \{x_{i1}, x_{i2}, ..., x_{im}\}$, *where* $V_i \neq \emptyset$, *further each* x_{ij} *has a dedicated state variable denoted as* $x_{ijstatus}$[2].
- *Further,* $V_{istart} \subseteq V_i$ *denotes the initial active variables of agent* a_i, *where* $V_{start} = \bigcup V_{istart}$ *and* $V_{start} \neq \emptyset$. *Further, the following condition must hold:* $\forall x_{ij}: (x_{ij} \in \{V_{istart}\}) \Rightarrow x_{ijstatus} = active$, *i.e.* $true \overset{ARV}{\Rightarrow} x_{ij}$.
- *A set of domains* $Doms = \{D_1, D_2, ...\}$, *where each variable* x_{ij} *is assigned to a domain* $D_l \in Doms$ *and* $Dom(x_{ijstatus}) = \{active, inactive\}$.
- *A set of constraints* C *distributed among agents, where* $C = C_C \cup C_A$ *and* C_{Cik} (C_{Aik}) *denotes compatibility (activity) constraint* k *of agent* a_i.

Find:

A solution Θ *representing an assignment to variables which meets the following criteria:*

1. *The variables and their assignments in* Θ *satisfy each constraint* $\in C$, *i.e.* Θ *is consistent with* $C_C \cup C_A$.
2. *All variables* $x_{ij} \in V_{start}$ *are active and instantiated.*
3. *There is no assignment* Θ' *satisfying 1. and 2., s.t.* $\Theta' \subset \Theta$.

Following this definition we give an example of representing a distributed configuration task as DDCSP *(Figure 1)*. A car manufacturer (a_1), a chassis and motor supplier (a_2), and an electric equipment supplier (a_3) cooperatively solve a distributed configuration task. Shared knowledge is represented through variables belonging to common constraints. User requirements are provided as additional constraints, which are denoted as C_R= {(package=standard),(car-body=4door-limo)}, further V_{1start} = {car-body, package}; V_{2start} = {transmission, motorization}; V_{3start} = {battery}. A solution for this configuration task is the following: {car-body= 4door-limo, package= standard, transmission= manual, motorization= 55bhp, battery= medium}. Note that the variables airbag, front-fog-lights, and electric-windows are not part of the above solution.

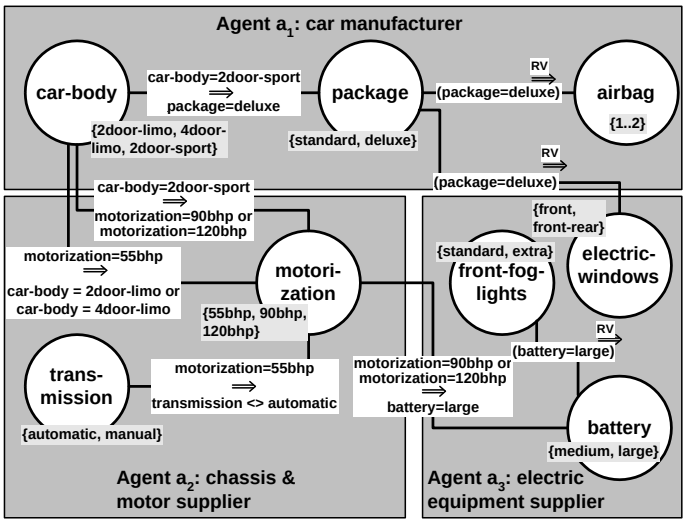


Figure 1. Example: Distributed Configuration

3 Solving Distributed Dynamic Constraint Satisfaction Problems

In the following we discuss our extensions to *asynchronous backtracking* [14], which is a complete, asynchronous version of a standard backtracking algorithm for solving distributed CSPs. Problem variables are distributed among problem solving agents, where each agent a_i has exactly two variables (x_i and $x_{istatus}$)[3]. Each agent variable x_i has a unique priority p_i, further x_i and $x_{istatus}$ always have the same priority[4]. We assume that constraints are binary, which is not limitating, because it is well known that any non-binary discrete CSP can be translated to an equivalent representation with

[2] In the following we denote x_{ij} as content variable, $x_{ijstatus}$ as status variable.

[3] The case of solving a distributed CSP with multiple local variables is discussed in [15]. In this case multiple virtual local agents (each working on a local variable) try to find a solution which satisfies all local constraints. The principles of asynchronous backtracking [14] remain the same for the case of multiple local variables.

[4] The lowest value represents the highest priority.

binary constraints. For this conversion two general methods are known: the dual graph method [6] and the hidden variable method [12]. We denote two agents whose variables participate in the same binary constraint as *connected*. Further we can say that these links between variables are *directed*, because of the unique priority that is associated with each variable. The agent with the higher priority is sending the variable value (via *ok?* messages) after each change of his variable instantiation to the connected agents. The latter evaluate their local constraints and inform corresponding agents about local inconsistencies (via *nogood* messages). *Nogoods* represent conflicting variable instantiations which are calculated by applying resolution. Their generation is a potential source of inefficiency because of space complexity[5]. Variable assignments of value sending agents are stored as tuple *(j,(y$_j$,d))* in the local *agent_ view* of the connected constraint evaluating agent, where y_j can either represent a content variable or a state variable.

3.1 Asynchronous Backtracking for DDCSP

Based on asynchronous backtracking we propose an algorithm for solving DDCSPs (*Algorithm 1*). The messages exchanged between agents have the following signatures, where j denotes the index of the message sending agent a_j.

- *ok?(j,(y$_j$,d)):* y_j denotes the variable name, and d the actual instantiation of y_j *(Algorithm 1 (a))*.
- *nogood(j,nogood$_j$):* nogood$_j$ is a set of inconsistent variable instantiations represented as *{(k,(y$_k$,d)),...}*, where k denotes the index of the agent a_k storing variable y_k with value d *(Algorithm 1 (b))*.

Both, *ok?*, and *nogood* messages can contain content information as well as state information about communicated variables, e.g. *ok?(1,(x$_{1status}$,inactive))* communicates activity information about variable x_1, *nogood(3,{(1, (x$_{1status}$,active)), (2,(x$_2$, 3))})* denotes the fact that either $x_{1status}$ must not be active or x_2 must not be instantiated with 3. If a tuple *(i,(x$_i$,d))* (x_i is a content variable) is added to the *agent_ view*, an additional tuple *(i,(x$_{istatus}$,active))* is added. If a tuple *(i,(x$_{istatus}$,inactive)* is added to the agent_view, the tuple *(i,(x$_i$,d))* is deleted from the *agent_ view*.

When the algorithm starts, all agents instantiate their active variables x_i, where $x_i \in V_{start}$, propagate their instantiations to connected agents and wait for messages. All other agents a_j propagate $x_{jstatus}$=*inactive* to connected agents[6]. When an agent receives a message, it checks the consistency of its local *agent_ view (Algorithm 1 (c))*. Similar to the ATMS-based approach discussed in [11] our algorithm contains *two* problem solving levels.

First, all local activity constraints are checked in order to determine the *Activity State Consistency*.

Definition 2 *Activity State Consistency*
Activity State Consistency is given, iff $x_{selfstatus}$ is consistent with agent_ view, i.e. all evaluated activity constraints are true under the value assignments of agent_ view, and all communicated nogoods are incompatible with agent_ view[7].

The variable $x_{selfstatus}$ represents the state variable of the local agent a_{self}[8]. The function *ASC (c.1)* checks the *Activity State Consistency*, tries to instantiate $x_{selfstatus}$ with a consistent value if needed, and returns *true* if *Activity State Consistency* is given, otherwise it returns *false*. If an inconsistent *Activity State* is detected, e.g. there is a contradiction between activity constraints, and $x_{selfstatus}$ can not be adapted consistently, *nogoods* are calculated including the domain constraints of $x_{selfstatus}$ and backtracking is done *(c.7)*. If *Activity State Consistency* is given and $x_{selfstatus}$ has been changed to inactive, an *ok?* message containing the new variable state is sent to the constraint evaluating agents *(c.6)*.

Second, if *Activity State Consistency* is given *(c.1)* and the local variable is active *(c.2)*, the algorithm checks the *Agent Consistency*.

Definition 3 *Agent Consistency*
Agent Consistency is given, iff the agent is in a Consistent Activity State and x_{self} is consistent with agent_ view, i.e. all evaluated compatibility constraints are true under the value assignments of agent_ view, and all communicated nogoods are incompatible with agent_ view.

The variable x_{self} represents the content variable of the local agent a_{self}. The function *AC (c.3)* checks the *Agent Consistency*, tries to instantiate x_{self} with a consistent value if needed, and returns *true* if *Agent Consistency* given, otherwise it returns *false*. If no *Agent Consistency* is given and x_{self} can not be instantiated consistently, *nogoods* are calculated including the domain constraints of x_{self} and $x_{selfstatus}$, and backtracking is done *(c.5)*. Else, if the value of x_{self} has been changed, an *ok?* message is sent to the connected constraint evaluating agents of a_{self} *(c.4)*.

Algorithm 1 *Asynchronous Backtracking for DDCSP*[9]

```
(a) when received (ok?(j,(yj,d))) do
      add {(j,(yj,d))} to agent_view;
      xselfstatusold ← xselfstatus; xselfold ← xself;
      check_agent_view; end do;

(b) when received (nogood(j,nogoodj)) do
      add nogoodj to nogood_list[10];
      if ∃ (k,(xk,d)) in nogoodj:
                  xk ¬connected[11] then
        request ak to add a link to self;
        add current {(k,(xk,d))} to agent_view;
      end if;
      xselfstatusold ← xselfstatus; xselfold ← xself;
      check_agent_view;
      if xselfold = xself ∧
            xselfstatusold = xselfstatus then
        send (ok?,(self,(xself,dself))) to aj;
      end if; end do;
```

[5] In *Section 3.3* we show how space complexity can be significantly reduced.
[6] This initialization is not included in *Algorithm 1*.
[7] *Agent_ view* is compatible with a *nogood*, *iff* all *nogood* variables have the same value as in *agent_ view*, $x_{selfstatus}$, and x_{self}.
[8] *self* denotes the index of the local agent a_{self}.
[9] The algorithm does not include a stable-state detection. In order to solve this task, stable state detection algorithms like distributed snapshots [2] are needed.
[10] The *nogood_ list* contains the locally stored *nogoods*.
[11] In order to check the *nogood*, all variables of connected agents part of the *nogood* must be represented in the *agent_ view*.

```
(c) procedure check_agent_view;
(c.1) if ASC(agent_view, x_selfstatus) then
(c.2)   if x_selfstatus = active then
(c.3)     if AC(agent_view, x_self) then
            if x_self ≠ x_selfold ∨
                 x_selfstatus ≠ x_selfstatusold then
(c.4)         send (ok?(self,(x_self, d_self)))
              to connected
                   constraint evaluating agents;
            end if;
(c.5)     else nogoods←{K_s|K_s ⊆ agent_view∧
           inconsistent(K_s,Dom(x_selfstatus))};
              backtrack(nogoods);
          end if;
        elseif x_selfstatus ≠ x_selfstatusold then
(c.6)      send (ok?(self,(x_selfstatus, d_selfstatus)))
           to connected
                constraint evaluating agents;
        end if;
(c.7) else nogoods←{K_s|K_s ⊆ agent_view∧
       inconsistent(K_s,Dom(x_selfstatus))};
                backtrack(nogoods);
      end if;
    end check_agent_view;

(d) procedure backtrack (nogoods);
      if ∅ ∈ nogoods then
        broadcast to other agents (¬∃ solution);
        terminate algorithm;
      end if;
      ∀ K_s ∈ nogoods do
        select a_k∈ agents(K_s):
                  lowest priority (a_k);
        send (nogood(self,K_s)) to a_k;
        remove {(k,(x_k, d)),(k,(x_kstatus, d_status))}
                    from agent_view;
      end do; check_agent_view;
    end backtrack;
```

3.2 Example: Solving a DDCSP

In the following we give a simple example consisting of three agents a_1, a_2, and a_3 (see *Figure 2*). Further we define a set of variables $\{x_1,x_2,x_3\}$ belonging to the agents a_1, a_2, and a_3 where $Dom(x_1)=\{1,2\}$, $Dom(x_2)=\{3\}$, and $Dom(x_3)=\{2\}$.

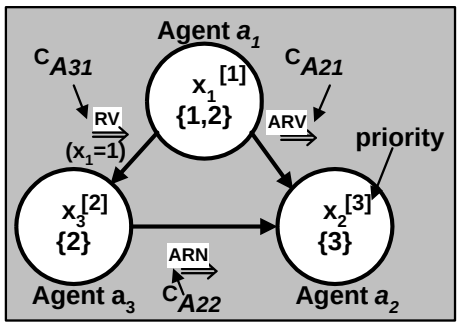


Figure 2. Example: Distributed Dynamic CSP

We define sets of initially active variables: $V_{1_{start}}=\{x_1\}$, $V_{2_{start}}=\emptyset$, $V_{3_{start}}=\emptyset$. In order to store the variable state of agent variables we define $\{x_{1status}, x_{2status}, x_{3status}\}$, where $Dom\ (x_{1status}) = Dom\ (x_{2status}) = Dom\ (x_{3status}) = \{active, inactive\}$. Finally, we introduce the following activity constraints: C_{A31}: $(x_1{=}1)\overset{RV}{\Rightarrow}x_3$, C_{A21}: $(x_1)\overset{ARV}{\Rightarrow}x_2$, and C_{A22}: $(x_3)\overset{ARN}{\Rightarrow}x_2$, where C_{Aik} denotes activity constraint k of a_i. *Figure 3* shows four snapshots of the solving process.

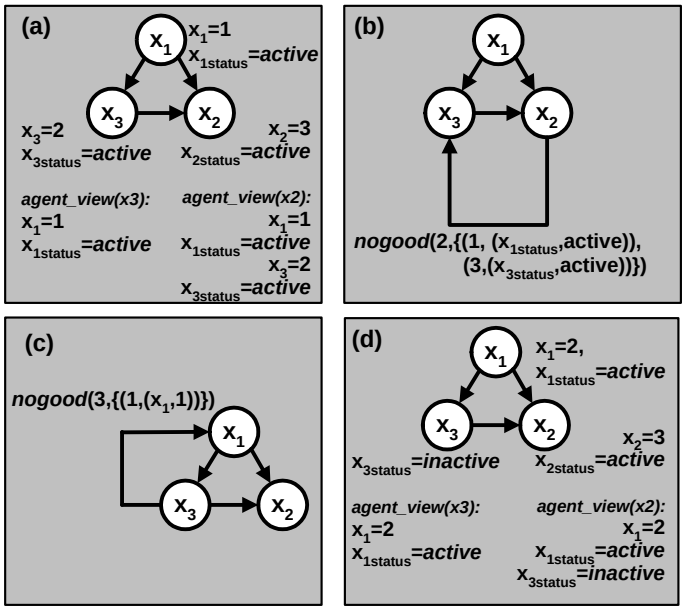


Figure 3. Views on the solving process

Agent a_1 locally instantiates its variables without regarding the instantiations of remote agents: $x_1{=}1$. Now a_1 sends its variable instantiation to the constraint evaluating agents a_2 and a_3 , i.e. *ok?(1,*$(x_1,1)$*)* . Agents a_2 and a_3 store these values in their local *agent_ view*. Both agent variables are activated, i.e. the state of x_2 and x_3 is changed to *active*. Now agent a_3 sends its variable instantiation to agent a_2, *i.e. ok?(3,*$(x_3,2)$*)*. The *view (a)* of *Figure 3* represents the situation after agent a_2 has stored the value change of x_3 in its *agent_ view*. Agent a_2 detects an inconsistency while checking its *Activity State Consistency*, since the activity constraints C_{A21} and C_{A22} are incompatible, i.e. x_2 can neither be activated nor deactivated. Agent a_2 determines those variables of the *agent_ view* responsible for the inconsistent *Activity State* and communicates *nogood(2,{(1,*$(x_{1status}$*,active)), (3,*$x_{3status}$*,active))})* to a_3, which is the lowest priority agent in the calculated *nogood (view (b) of Figure 3)*. This *nogood* is stored in the *nogoods_ list* of a_3. Agent a_3 detects an inconsistent *Activity State*, since no value of $x_{selfstatus}$ is consistent with the *agent_ view* and C_{A31}. The *nogood {(1,*$(x_1,1)$*)}* is sent to a_1 *(view (c) of Figure 3)*. Note that *(*x_1*,1)* $\Rightarrow$ *(*$x_{1status}$*,active)*.

Agent a_1 is in a consistent *Activity State*, i.e. no activity constraints are violated. For achieving *Agent Consistency* the value of x_1 is changed, i.e. $x_1{=}2$. The new instantiation is propagated to a_2 and a_3. Subsequently a_2 and a_3 enforce *Activity State Consistency*. x_3 is deactivated *and ok?(3,(*$x_{3status}$*,inactive))* is communicated to a_2. Agent a_2 removes *(3,*$(x_3,2)$*)* from its *agent_ view (view (d) of Figure 3)*.

3.3 Analysis

In order to show the soundness of *Algorithm 1* we must show that each generated solution (algorithm is in a stable state, i.e. all agents wait for incoming messages and no message is sent) satisfies the criteria stated in *Section 2. First*, the assignments satisfy each constraint, since each agent checks all local constraints after each value change in the local *agent_ view*.

Second, all variables $x_i \in V_{start}$ are active and instantiated in a solution, since $x_{istatus}$=*active* is a local constraint of agent a_i and all active variables are instantiated. Further, x_i can not be deactivated, since $x_{istatus}$=*active* holds. *Third*, the minimality of the solution is guaranteed, since no variable has a solution relevant value, unless it is *active*. Each variable has well founded support either through $\overset{(A)RV}{\Rightarrow}$ constraints or through its membership in V_{start}. After each change in the local *agent_view*, the activity state of the variable is checked and updated. If no $\overset{(A)RV}{\Rightarrow}$ constraint is activated, the variables state is set to *inactive*, i.e. there is no further well founded support for the variable.

In order to show the completeness of the algorithm we must show that if a solution exists the algorithm reaches a stable state, otherwise the algorithm terminates with a failure indication (empty *nogood*). Let us first assume that the algorithm terminates. If this algorithm terminates by reaching a stable state then we have already shown that this is a correct solution. If this algorithm terminates by deducing the empty *nogood* then by applying resolution we detected that no consistent value assignment to the variables exists, i.e. no solution to the DDCSP exists. Finally we have to show that the algorithm terminates. Sources for infinite processing loops are cycles in message passing and subsequent searching of the same search space. Infinite processing loops are avoided because we require a total order of the agents. The *ok? (nogood)* messages are passed only from agents to connected agents with lower (higher) priority. *Nogoods* avoid subsequent searching of the same search space.

Dynamic constraint satisfaction is an NP-complete problem [13]. The worst-case time complexity of the presented algorithm is exponential in the number of variables. The worst-case space complexity depends on the strategy we employ to handle *nogoods*. The options range from unrestricted learning (e.g. storing all *nogoods*) to the case where *nogood* recording is limited as much as possible. For the agents in the presented algorithm it is sufficient to store only one *nogood* for each $d \in Dom(x_i)$. These *nogoods* are needed to avoid subsequent assignment of the same value for the same search space. In addition all *nogoods* can be deleted which contain a variable-value pair not appearing in the *agent_view*. Consequently the space-complexity for *nogood* recording is $O(n\ |D|)$, where n is the number of agents and |D| is the maximum cardinality of the domains. In addition the generation of *nogoods* is another source of high computational costs. Note, that in the presented algorithm it is sufficient to generate one *nogood*. This *nogood* needs not be minimal, i.e. for the *nogood* generation in the procedure *backtrack* even the complete *agent_view* is an acceptable *nogood*. However, non-minimal *nogoods* lead to higher search efforts. The advantages and strategies for exploiting *nogoods* to limit the search activities are discussed in [1], [5].

4 Related Work

Different algorithms have been proposed for solving distributed CSPs. In [10] a distributed backtracking algorithm (DIBT) is presented which is based on the concept of graph based backjumping. The exploitation of *nogoods* is not supported. DIBT is especially powerful if combined with variable ordering techniques. Note that our presented algorithm can use any total ordering, which takes advantage of the actual problem structure and this algorithm also performs graph based backtracking. However, in practice variable ordering must take into account that sets of variables are assigned to one agent and that this assignment cannot be changed because of security or privacy concerns.

Asynchronous weak commitment search proposed by [14] employs a min-conflict heuristic, where a partial solution is extended by adding additional variables until a complete solution is found. In contrast to asynchronous backtracking all detected *nogoods* must be stored, in order to prevent infinite processing loops. In many configuration domains the problem size does not permit the storage of all generated *nogoods*, i.e. asynchronous backtracking is more applicable.

In [4] an agent architecture for solving distributed configuration-design problems employing an algorithm based on the concurrent engineering design paradigm is proposed. The whole problem is decomposed into sub-problems of manageable size which are solved by agents. The primary goal of this approach is efficient distributed design problem solving, whereas our concern is to provide effective support of distributed configuration problem solving, where knowledge is distributed between different agents having a restricted view on the whole configuration process.

5 Conclusions and Further Work

The integration of businesses by internet technologies boosts the demand for distributed problem solving. In particular in knowledge-based configuration we have to move from standalone configurators to distributed configuration. Dynamic constraint satisfaction is one of the most applied techniques in the configuration domain and therefore we have to extend this technique to distributed dynamic constraint satisfaction. In this paper we proposed a definition for distributed dynamic constraint satisfaction. Based on this definition we presented a complete and sound algorithm. This algorithm allows the exploitation of bounded learning algorithms. Configuration agents can exchange information about conflicting requirements (e.g. *nogoods*) thus reducing search efforts. The algorithm was implemented by using ILOG configuration and constraint solving libraries. The concepts presented in this paper are an essential part of an integrated environment for the development of cooperative configuration agents, where a conceptual model of configuration agents is automatically translated into an executable logic representation [8].

Further work will include additional applications of the presented algorithm to various configuration problems. These applications will help us to gain more insights in the nature of configuration problems thus providing the basis for further work on DDCSP strategies. E.g. applications in the telecommunication domain support the assumption that in configuration domains *nogoods* tend to be small in their arity. This suggests the application of constraint learning techniques to focus the search.

In addition we are investigating extensions of the basic dynamic CSP paradigm in order to include concepts such as disjunction or default negation as proposed by [13] and generative CSP representation [9].

REFERENCES

[1] R.J. Bayardo and D.P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings AAAI*, pages 298–304, Portland, Oregon, 1996.

[2] M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3,1:63–75, 1985.

[3] R. Weigel D. Sabin. Product Configuration Frameworks - A Survey. In E. Freuder B. Faltings, editor, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13,4, pages 50–58. 1998.

[4] T.P. Darr and W.P. Birmingham. An Attribute-Space Representation and Algorithm for Concurrent Engineering. *AIEDAM*, 10,1:21–35, 1996.

[5] R. Dechter. Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence*, 40,3:273–312, 1990.

[6] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[7] B. Faltings, E. Freuder, and G. Friedrich, editors. Workshop on Configuration. *AAAI Technical Report WS-99-05*, Orlando, Florida, 1999.

[8] A. Felfernig, G. Friedrich, and D. Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. In *11th International Conference on Software Engineering and Knowledge Engineering*, pages 337–345, Kaiserslautern, Germany, 1999.

[9] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. In E. Freuder B. Faltings, editor, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13,4, pages 59–68. 1998.

[10] Y. Hamadi, C. Bessiere, and J. Quinqueton. Backtracking in distributed Constraint Networks. In *Proceedings of ECAI 1998*, pages 219–223, Brighton, UK, 1998.

[11] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proceedings of AAAI 1990*, pages 25–32, Boston, MA, 1990.

[12] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of ECAI 1990*, Stockholm, Sweden, 1990.

[13] T. Soininen, E. Gelle, and I. Niemelä. A Fixpoint Definition of Dynamic Constraint Satisfaction. In *5th International Conference on Principles and Practice of Constraint Programming - CP'99*, pages 419–433, Alexandria, USA, 1999.

[14] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem. *IEEE Transactions on Knowledge and Data Engineering*, 10,5:673–685, 1998.

[15] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS-98), Paris*, pages 372–379, 1998.

User Interface Requirements for Knowledge Acquisition and Modeling

Gerhard Fleischanderl [1]

Abstract. User interfaces for knowledge acquisition are mainly targeted at expert users or knowledgeable intermittent users. Therefore the design of such a user interfaces has to find the balance between ease-of-use and handling-efficiency. Out of a set of general rules for interface design, enabling shortcuts, informative feedback and error prevention seem to be most important for knowledge acquisition tasks.

1 INTRODUCTION

Within this paper, knowledge acquisition means knowledge base modeling, i.e. editing and maintaining knowledge bases.

The user interfaces for knowledge acquisition must fulfil particular requirements that are different from "normal" user interfaces. Knowledge acquisition is similar to programming and is often done by computer and software specialists. Furthermore, domain experts also do knowledge acquisition tasks.

This short paper contrasts the general ergonomic principles for user interfaces [1] with the requirements for knowledge acquisition. Three real-world user interfaces for knowledge acquisition for configurators are compared to the general principles.

2 EXAMPLES FROM REAL-WORLD PROJECTS

This paper draws on experience from several configurators [2] and knowledge-acquisition interfaces that are in production use in Siemens sales and engineering departments. From these real-world applications we pick two examples to illustrate the requirements and discuss the experiences from usage in the field. The two examples cover a relevant range of user interfaces. Furthermore, we discuss properties of the visual modeler described in [3].

KBE - Knowledge Base Editor for configurable products

KBE is used to model the configuration knowledge about products that are sold with private branch exchanges (e.g. phone devices, specialized software packages). The knowledge bases edited with KBE are then compiled into code for different configurators.

In a structured tree view, the user can find and access the configuration objects of he knowledge base. The properties of an object can be manipulated in a detailed input form. In particular, KBE supports the editing of expressions in an efficient manner.

KBE was specifically designed to make the editing and maintenance of configuration knowledge bases easier and faster, and to improve the quality of the knowledge bases. The features of KBE include checks for unused attributes, a syntax-driven formula editor, undo-redo facility, and an update mechanism for importing data from external product databases.

Engineers from the author's team developed KBE.

CBC - Class browser with constraints

For editing the knowledge base for a configurator for telecom switches, we used the class browser of the underlying Smalltalk environment and added a facility for defining constraints.

The class browser has selection windows for accessing classes, their attributes and methods. In a separate input window the properties of a class can be manipulated.

CBC offers advantages to programmers who are familiar with development environments for object-oriented programming languages. Furthermore, we introduced tables to allow non-expert users to do routine maintenance like modifications to part numbers.

CBC was developed within a larger project led by the author.

Visual modeler

In [3], Heiderscheidt and Skovgaard describe their approach for the visualization of configurations and a visual modeler. Product model rules and visual rules are handled separately from one another. In that environment the user can see both the configuration model and visual model. Checks between the models assist the user when he/she creates the rules for visualization.

3 USAGE PROFILES

We distinguish three generic groups in a user community [1].

Novice or first-time users

This user group is rare in knowledge acquisition tasks. For knowledge acquisition, users really should have some training or relevant experience. Untrained users may do simple tasks, like modifications to part numbers.

[1] Siemens AG Österreich, Program and Systems Engineering,
Erdberger Laende 26, A-1030 Vienna, Austria,
email: gerhard.fleischanderl@siemens.at

Knowledgeable intermittent users

As knowledge-based systems are used for more applications, we will find more users with some skills in using a knowledge-acquisition user interface. They will understand some of the concepts, but will need support for finding their way around the features of the user interface. For them the user interface ought to be verbose.

Expert frequent users

The 'power' users want to do their tasks quickly and demand an efficient user interface.

Comparison

For knowledge acquisition, expert frequent users were the main target group. Yet, intermittent users will become more and more important as users.

The race continues between improving the user interfaces for knowledge acquisition and enlarging the capabilities of the underlying knowledge-based systems. More features mean more variety. Therefore the frequent users, too, become intermittent users when they come across a feature they rarely use.

KBE was designed for expert users as well as intermittent users. Therefore, its features are a compromise between the requirements of the two user groups.

CBC is targeting the expert users. It is hard to learn and hard to use for non-experts.

The visual modeler described in [3] provides an integrated environment for modeling both product rules and visualization rules. This helps the intermittent users to work more efficiently.

4 THE EIGHT GOLDEN RULES OF INTERFACE DESIGN

Shneiderman [1] presents eight rules that are the underlying principles of good interface design. In this section we describe the eight rules and compare them each to properties of the interfaces KBE and CBC. Finally, we subjectively rate each rule whether it is important for knowledge acquisition.

Rule 1: Strive for consistency

For instance, consistent sequences of actions should be required in similar situations. Yet, there are many forms of consistency, which often prevents overall consistency.

- KBE uses a tree view and input forms for letting the user edit a knowledge base. The input forms contain sub-windows for clustering data. This layout is used for all windows.
- For editing constraint code, CBC allows input via forms or direct source-code input. Input forms for tables are again different from the input forms for constraints. The mixture between forms and pure source-code is sometimes helpful, sometimes a bit disturbing. Using forms (input fields) on top of a standard class browser creates inconsistency for the user.
- In [3] a visual modeler is described. There the product model rules and visual rules are separated, which reduces complexity for maintenance. Knowledge acquisition for product model and visual rules, respectively, are consistent each among themselves.

Relevance of this rule to knowledge acquisition: *Medium*. Lack of consistency makes the handling more tedious, but users will partly get accustomed to that.

Rule 2: Enable frequent users to use shortcuts

Frequent users often appreciate abbreviations and macro facilities.

- KBE allows the user to define abbreviations (macros) that can be used in expressions.
- CBC does not support shortcuts other than those found in the standard class browser.

Relevance of this rule to knowledge acquisition: *High*. Speed is important, especially if the expert users can envisage ways for accelerating their work.

Rule 3: Offer informative feedback

For every user action, there should be system feedback. The visual presentation (graphical display) offers better feedback than text-based interfaces.

- KBE checks whether parameter names that are not defined anywhere are used in rules or formulas. This may happen in particular when an existing parameter is removed, but was used within an expression.
- CBC checks the syntax and other properties of the underlying programming language. This is done after having completed an input sequence.

Relevance of this rule to knowledge acquisition: *High*. Poor feedback annoys and hampers every user every time.

Rule 4: Design dialogs to yield closure

The completion of a group of actions should be highlighted to the user.

- KBE does not use explicit closure of a series of actions. Explicit closure would make every action longer, thus annoying the experienced users. There is an automatic "to do list", namely the flags for errors (e.g. undefined parameters). The user knows that every error flag has to be dealt with and can easily find out whether this job is completed.
- CBC only uses features of the standard class browser.

Relevance of this rule to knowledge acquisition: *Medium*. Users can compensate for missing closure.

Rule 5: Offer error prevention and simple error handling

As much as possible, design the system such that users cannot make a serious error. If users make an error, the system should detect the error and offer constructive instructions for recovery.

- KBE has a syntax-driven formula editor that allows only formulas that have correct syntax. Furthermore, parameters must have been declared before using them in formulas.
- CBC only uses features of the standard class browser.
- In [3] the product model and the visual model are separated. Yet, there are update mechanisms and checks in place to alert the user when changes were done in any one model.

Relevance of this rule to knowledge acquisition: *High*. Preventing mistakes in the first place is an excellent way to improve efficiency and satisfaction.

Rule 6: Permit easy reversal of actions

- KBE allows chronological undo and redo across multiple steps of actions.
- CBC uses the features of the standard class browser, which includes undo and redo of the last action.

Relevance of this rule to knowledge acquisition: *Low* (but desirable). Often a user interface offers features for 'manual reversal' of actions, e.g. insert and delete.

Rule 7: Support internal locus of control

Experienced users strongly desire the sense that they are in charge of the system and that the system responds to their actions.

- KBE guides the user and does checks, but does not force him/her into particular sequences of actions. At the end, source code for different configurators is generated. The users see the automatic generation as a big advantage and not as loss of control.
- When working with CBC, the user has to be in charge (to put it in positive words). In this case, this is only good for very experienced users who may nevertheless prefer working in a plain text file. So CBC looks like a compromise that does not really please anybody.

Relevance of this rule to knowledge acquisition: *Low*.

Rule 8: Reduce short-term memory load

The user interface should relieve the user from memorizing previous data to carry out an action.

- In KBE, input fields in forms and sub-windows are clustered according to whether they belong together when specifying the contents of a knowledge base. With navigation between parts of the tree view and sub-windows, KBE supports the search for information outside the current input focus.
- CBC does not support the search for distributed information very well. This is more easily achieved by editing the knowledge base as one (large) text file.
- In [3] the user can see both the configuration model and the visual model. This enables easy access to the information needed for creating the visualization rules.

Relevance of this rule to knowledge acquisition: *Medium* (but hard to achieve for complex modifications). Being able to carry out a task without switching between windows is desirable. Yet, the user interface should not get overloaded with information.

5 CONCLUSION

The principles for good interface design also apply to user interfaces for knowledge acquisition. Yet, the requirements of knowledge acquisition interfaces emphasize the needs of frequent users.

- KBE: The effort of designing KBE for usability and efficiency paid off. The experiences of both expert and intermittent users are very favorable towards KBE.
- CBC: In retrospect, maintenance of the knowledge base was done infrequently and relied heavily on reusing code and modifying it to new requirements. The standard class browser lacks the facilities to easily find all the occurrences of strings and sub-strings. It offers too much structure and too little flexibility for finding and reusing ‘similar’ code.

  As it turned out, the knowledge base most of the time was maintained by editing the plain text file and loading the file into the configurator. Having the syntax only checked when loading the text file is only a minor drawback for expert users. Yet, non-expert users can only do easy tasks with CBC.
- The visual modeler and integrated environment described in [3] addresses very well some principles for interface design.

When designing a user interface for knowledge acquisition, the designers should find out who the users would be. In general, the designers then ought to favor the rules no. 2 (enable shortcuts), no. 3 (give informative feedback) and no. 5 (prevent errors).

Beyond the general ergonomic principles discussed in this paper, specific requirements and guidelines for designing knowledge base modeling tools can be established. This has to take into account the environment and the user groups of the tool. Usability tests can be done to find out the user interface design principles that are relevant to the task.

REFERENCES

[1] B. Shneiderman, *Designing the User Interface (Strategies for Effective Human-Computer Interaction)*, Addison Wesley Longman, 3rd edn, 1998.

[2] G. Fleischanderl, G. E. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner, 'Configuring Large Systems Using Generative Constraint Satisfaction', *IEEE Intelligent Systems & their applications*, 13:4, pp.59-68, July/Aug. 1998.

[3] D. Heiderscheidt and H.J. Skovgaard, 'Visualization of Configurations: Simplifying Configuration User Interfaces', *AAAI Technical Report WS-99-05*, pp.114-118, Papers from the AAAI Workshop on Configuration, Orlando, Fla., July 1999.

Configuration of a machining operation

Laurent Geneste[1], **Magali Ruet**[1], **Thibaud Monteiro**[2]

Abstract. The problem of configuring a machining operation is complex (many parameters and many interactions between parameters) and is generally achieved thanks to expert heuristic knowledge such as sequential choice of the parameters according to local technical constraints or reuse and adaptation of an already defined solution. In order to support the configuration of a machining operation, we describe first the nature of the data involved and then the different solving processes that can be used.

Keywords. Machining operation configuration, C.S.P., C.B.R.

1 INTRODUCTION

In recent research work, the need of interactive decision support system in the field of product configuration has been clearly emphasised for example in [1]. However, the problem of machining operation configuration is scarcely addressed in the literature. This paper aims at presenting the specificity of this configuration problem and of the solving process.

The problem of configuration of a machining operation, illustrated by figure 1, may be described as follows: in order to ensure an adequate machining of a part, several parameters for the machining operation have to be defined. These parameters are for instance the kind of operation that will be performed (e.g. milling, drilling…), the machine on which the operation is to be performed, the tools that will be used, the feed rate and so on.

The choice of the parameters must be achieved in accordance with technical constraints (some tools do not adapt on some machines, the energy necessary for the operation should be available on the machine…). Moreover, optimisation criteria are often defined in order to select a solution among the set of configurations that respect the technical constraints.
This problem is very complex essentially because of the combinatorial characteristics of the domain of search. The complexity of the problem leads experts who carry out machining process configuration to follow two different heuristic methods :

- sequential selection of the parameters and therefore progressive building of a solution without taking into account the whole set of technical constraints,
- reuse and adaptation of an already defined solution.

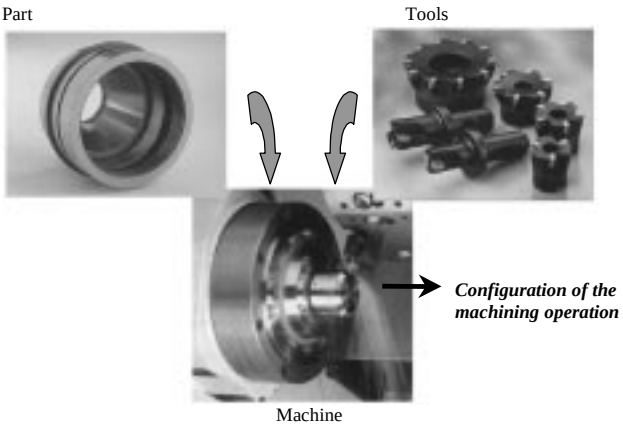


Figure 1. Configuring a machining operation

Several decision support tools are provided, especially by tool manufacturers. These tools enable to select easily some of the parameters according to a specification of the context. For example given the characteristics of a tool and wear and of the part, such a system may compute a relevant value for the feed rate and the depth of cut in a turning operation. However, once again, these tools are used in a specific step of the configuration process, when the major part of parameters is already defined.

As a conclusion, we observe that the problem of machining process configuration remains most of the time based on empirical data layout and methodologies. This leads us to propose an explicit representation of the domain knowledge (knowledge layout, decision variables and technical constraints) and a set of manipulation techniques that support the configuration process in order to provide the user with an interactive decision support system that allow him to develop his own strategies for the problem solving.

[1] Equipe PA – LGP - ENIT - Avenue Azereix, 65000 Tarbes, France, email: laurent@enit.fr, ruet@enit.fr
[2] LAG - ENSIEG - INPG - Rue de la Houille Blanche - B.P. 46, 38402 Saint Martin d'Hères Cedex – email: Thibaud.Monteiro@inpg.fr

2 CONFIGURATION DATA MODELLING

In order to make a standard modelling of the domain knowledge, we selected the object oriented modelling language U.M.L. (Unified Modelling Language) [2]. We use the U.M.L. class diagram and object diagram in order to represent the domain data. In order to explicit the technical constraints, we added to the basic U.M.L. graphic notation, a specific notation for the concept of constraint as illustrated by figure 2. We can observe that the classical concept of association is a specific case of constraint in which the involved attributes are the references of the associated objects.

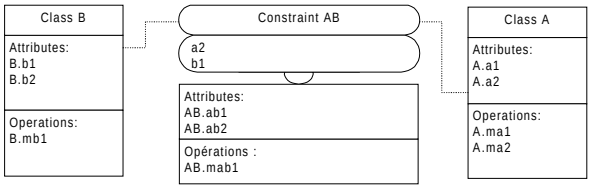


Figure 2. Notation of a constraint between two attributes

According to [3] and to several experts of the manufacturing domain, we identified a set of variables (attributes of the classes of the U.M.L. class diagram) and of constraints that act upon the configuration of a machining operation [4].

We identified more than 30 variables that are useful for the configuration of a machining operation. Some of this variables are defined over discrete domains (for instance **X4**: tool holding device, **X7**: type of insert or **X19**: tool material) and the others are defined over continuous domains (for instance ***X8:*** precision , ***X14***: cutting speed or ***X22***: surface roughness of part).

We also identified 32 technical constraints that should be taken into account in order to make an adequate configuration of the machining operation.

We can observe that some constraints are:

- binary such as:
  - **C1** (X5,X7) : Adaptation constraint between the kind of tool shank and the type of insert. Cutting tool manufacturers generally provide with a list of valid couples (tool shank, wear))
- of arity more than 2 such as:
  - **C11**(X7,X10,X11,X14,X16,X20) : The chip breaker diagram: for a good machining , it is important that the chip is fragmented. The fragmentation of the chip for a given cutting speed Vc and feed rate f. We obtain experimentally a working zone where the couple (Vc,f) leads to a broken chip.*).*

Some constraints are on variable defined :

- over discrete domains only as for instance:
  - **C2**(X6,X7) and **C3**(X5,X6) : There exist four normalised clamping systems. The choice of a clamping system is linked to the choice of the insert (C2) and of the tool shank (C3),
- over continuous domains only as for instance:
  - **C23**(X11,X24,X26) : The effective length of cut L is function of the working major cut edge angle κr and of the depth of cut.
  - **C31**(X9,X24,X31) : The feed force Ff is function of the cut force Fv and the direction of the major cutting edge angle κr.
- over both discrete and continuous domains as for instance:
  - **C8**(X6,X9) : The cut force must be tolerable for the cutting tool. Over a given value of cutting force the tool begins to bend and the result of the machining operation may not be satisfying.
  - **C17**(X10,X21,X22) : The surface roughness of the part is function of the machining conditions, of the corner radius Rε and of the feed rate f of the tool.

Some constraints are described

- in extension (table) such as:
  - **C9**(X1,X5) : In order to machine a profile, only some tool shank are acceptable. Indeed, it is necessary that the shape of the tool is acceptable for the profile to machine. Manufacturers provide with a list of couples (tool shank, profile).
- in intention (predicate) such as:
  - **C31**(X9,X24,X31) : The feed force Ff is function of the cut force Fv and the direction of the major cutting edge angle κr.

$$Ff = Fv.(0{,}15 - 0{,}1.\cos\ \chi_r)$$

3 CONFIGURATION PROCESS

As exposed in part 1, two major approaches are used when configuring a machining operation.

3.1 Configuration as a C.S.P.

The first approach is to build incrementally a solution by restricting the domain of the variables according to a set of technical constraints and try to use the expert knowledge for guiding the search first to a solution and hopefully to a good one. A substantial decision support would be to guide the search of the solution by interactively applying a set of constraints chosen by the machining operation designer. In this case, the problem of configuration can be compared to a Constraint Satisfaction Problem and all the techniques developed in this field can be applied. We use the DnGAC-4 [5] algorithm for a dynamic (one can add/remove constraints dynamically) propagation of constraints of arity n defined on discrete domains only. For constraints of arity n defined on continuous domains only, recent approaches [6] provide a relevant framework for a generic constraint propagation based on a decomposition of the constraints into quadtrees (for binary constraints) or more generally 2^k trees (for constraints of arity superior to 2).

A recent work [7] has emphasised the need of interactive configuration tools taking into account both discrete and continuous constraints for example in the field of Computer Aided Design. This works points out and explains the relevance of a constraint based approach for configuration compared to the rule based approach. Two kinds of constraints are depicted:

- activity constraints [8] which enable an incremental introduction of the variables and of the constraints,

- compatibility constraints which are the constraints of the domain.

However, many constraints that are to be manipulated have both discrete and continuous variables. In this case, the preceding techniques can not be applied directly and we are looking for a way to manage this kind of constraints. Another interesting point is that within the object oriented modelling some implicit constraints are present and could be used to accelerate the solving process. For example, when the designer defines a class, he defines an implicit constraint between the attributes of the class, and this constraint could be propagated.

3.2. Configuration as a C.B.R.

The second approach is to find a solution that has already been validated on a similar case and to adapt it for solving the new problem. In order to support this process, we provide the operation designer with a fuzzy C.B.R. (Case Based Reasoning) mechanism [9], [10]. The basic algorithm is recursive and propagates through the object structure. The user can weight the attributes in order to describe their expected influence on the result of the comparison. The result of the search is a necessity and a possibility degree that represent to which point a case stored in the object structure is close to the case to be solved.

Again, the underlying object structure is used in order to carry out a rough filtering of the potential objects of interest in the object structure and then to retrieve specific relevant cases.

3.2.1. *Rough filtering*

In order to enable a fast selection of the objects potentially interesting in the similar case search, we propose to exploit the object modelling as an indexing base. The filtering process is therefore achieved by comparing the characteristics of the class of the reference object (o belonging to class O) and each class of the class diagram of the domain (class O'). From the similarity between classes O and O', we can decide whether or not to inspect in more detail the objects belonging to class O'. We use the following notation to describe the used algorithm:

name(a) : name of the attribute a

class(a): class of the attribute a

value(a): value of the attribute a

w_a: subjective weight of attribute a

Rough filtering algorithm: computing of S(O,O')

Initialisation

$S=0$

For each attribute a belonging to class O

$S = S + w_a$ if name(a) belongs to class O'

For each aggregation C belonging to class O (O is composed of C)

$S = S + w_C.S(C,C')$

Normalisation

$$S = S / \sum w_{i,i \in O}$$

Let us notice that the weights attributed by the user to the attributes are used in order to describe the reference case are used by the rough filtering algorithm. This enables, when required, to find similarities between very different classes which have in common attributes important for the seach of the user.

3.2.2 *Case retrieval*

Once the set of candidate classes is selected, we can achieve the search of cases similar to the reference case among the objets derived from these classes. The following algorithm computes the necessity of similarity of objects o and o'. A similar algorithm enable to compute the corresponding possibility degree.

Case retrieval algorithm: computing of N(o,o')

Initialisation

N = 1

For each aggregation contained in object o (os is composed of c, where c is an instance of class C), three cases are possible:

Case 1 : o' is composed of c', where name(c') belongs to class C

$$N = \min(\max(1-w_C, N(c,c')),N)$$

Case 2 : o' is composed of an object which name belongs to class C but not instanced

$$N = \min(1-w_C, N)$$

Case 3 : o' is not composed of an object which name belongs to class C

$$N = \min(1-w_C, N)$$

For each elementary attribute a belonging to object o, three cases are possible:

Case 1 : if the corresponding attribute a' exists in object o' and has a value

$$N = \min(\max(1-w_a, N_a(a,a')),N)$$

Case 2 : if the corresponding attribute a' exists in object o' but has no value

$$N = \min(1-w_a ,N)$$

Case 3 : if the corresponding attribute does not exist in o'

$$N = \min(1-w_a, N)$$

3.3 Combining C.S.P. and C.B.R.

In fact, we think that both approaches may be complementary and could be used together to support the configuration process.

First, the constraint propagation techniques may enable to restrict rapidly the domain of search of the case based reasoning algorithm. Combined to the rough filtering algorithm, it can make the search easier by restricting the domain of search according to the technical constraints and to the preferences of the user.

Another way to combine both techniques is to used the constraint propagation when the user wants to adapt an existing solution in order to solve a new problem. We can whether propose to the user a neighbourhood of the retrieved case that respect the constraints or let the user define the adaptation by checking the constraint satisfaction at each step.

4 IMPLEMENTATION

We propose an integrated framework called KASKOO (Knowledge Acquisition System for Object Oriented Knowledge)for an interactive decision support system generator based on an object oriented knowledge representation and acquisition module and a methodology of knowledge acquisition. The acquired knowledge will then be used thanks to various and generic manipulation techniques, such as constraints problem solving and / or case based reasoning.

When a large amount of information is to be used by a decision support system, in order to ensure several desirable properties such as maintainability, reliability and so on, the corresponding knowledge should be structured. Moreover, a knowledge acquisition methodology must be available in order to help the user to organise his knowledge in a relevant structure. This section describes such a methodology and the corresponding tools provided by the KASKOO environment.

The methodology for knowledge acquisition in KASKOO may be divided into several major steps:

4.1 Specification of the knowledge structure.

In KASKOO, in accordance with the standard concepts of classes and objects, two levels of representation are defined: the class level and the object level. These levels are successively developed in paragraphs 3.2 and 3.3.

4.2 Implementation of the class structure

The class level enables to implement the class structure. Classical object oriented programming languages, such as C++, do not provide an easy evolution of the knowledge structure. For example, when an attribute has top be added in a class of objects, the code of the class must be updated as well as the user interface and the database model (if needed) and the code must be recompiled. These operation often require specific skills in programming that the user of a decision support system in the field of manufacturing does generally not master.

Moreover, in order to build a knowledge based system with such an environment, specific developments for the graphical user interface and to ensure object persistence are required. That is why we propose a dynamic object oriented knowledge acquisition system with the following properties:

- Basic object oriented concepts: attributes and methods, inheritance, association, aggregation,
- Constraints between class attributes,
- Dynamic structure of the classes: the acquisition system enables a dynamic implementation of the classes and objects. In KASKOO, all features may be Created, Modified or Deleted as well in the model as in the instances.

4.3 Implementation of the objects of the domain

The object level enables to instanciate the class structure with the appropriate objects. In order to facilitate the knowledge capitalisation, several acquisition mechanisms have been provided.

- Automatic generation of a Graphical User Interface (GUI). The Graphical User Interface for knowledge acquisition in KASKOO is automatically derived from the class structure. This enables to use a homogeneous presentation of the interface that facilitates the user understanding. Moreover, changes in the shape of the GUI are factorised and dynamically applied to the whole GUI.
- Automatic generation of persistent database for objects. In order to store the knowledge, a generic persistence mechanism is used. This feature relies upon an object oriented database. The drivers that connect transient (non persistent) objects and their persistent image are generated automatically by KASKOO.
- Dynamic structure objects. The acquisition system enables a dynamic implementation of the objects. In KASKOO, all features may be Created, Modified or Deleted as well in the class model as in the object model.

The automatic generation of GUI and persistence provides the user with a standard interface representation and enables him to focus on the central problem of knowledge modelling without been disturbed by the computer technique.

5 CONCLUSION

In this paper we describe the problem of configuring a machining operation. This problem leads first to the need of domain representation and to do so, we propose to enrich an existing object oriented modelling language (U.M.L.) with an explicit description of the notion of constraint. Then, we define two different ways to use the knowledge in order to carry out the configuration process. The first way is to use constraint propagation techniques in order to restrict the domain of search of a valid configuration. The second way is to use previously defined solution and to adapt them to a new situation. We propose some ideas on the way to combine both approaches.

6 REFERENCES

[1] Brown D.C., Some Thoughts on Configuration Processes, *AAAI 1996 Fall Symposium Workshop: Configuration*, MIT, Cambridge, Massachusetts, USA, November 9-11, 1996.

[2] Fowler M., Scott K., Booch G., *Uml Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley Pub Co, 1997.

[3] Boothroyd G., Knight W.A., *Fundamentals of machining and machine tools*, Marcel Dekker Inc., 1989.

[4] Monteiro T, Perpen J.L., Geneste L. Configuring a machining operation as a constraint satisfaction problem. *In nternational Conference on Computational Intelligence for Modelling Control and Automation*, Vienne : Austria, 1999.

[5] Bessière C., Arc-consistency in dynamic constraint satisfaction problems. *Proceedings of the 10th AAAI*, California, p. 221-226, 1991.

[6] Sam D., Constraint consistency techniques for continuous domains, PhD Thesis, EPFL Lausanne, 1995.

[7] Gelle E., Weigel R., Interactive Configuration using Constraint Satisfaction Techniques, *Second International Conference on Practical Application of*

Constraint Technology, PACT-96, pp57-72, London, April 1996.
[8] Mittal S., Falkenhainer B., Dynamic Constraint Satisfaction Problems, *Proceedings of AAI-90*, pp25-32, 1990.
[9] Kolodner J., *Case Based Reasoning*, Morgan Kaufman Publishers Inc., 1993.
[10] Aamodt A., Plaza E., Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *Artificial Intelligence Communications*, IOS Press, vol. 7 : 1, pp. 39-59. 1994.

Description and Configuration of Complex Technical Products in a Virtual Store

Diego Magro and Pietro Torasso
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185; 10149 Torino; Italy
{magro, torasso}@di.unito.it

Abstract. The paper describes a representation formalism for the description of products and the associated reasoning mechanisms for the configuration of complex products in a virtual store where a customer (potentially non expert of the domain) has to be supported in the task of selecting and assembling a product. We present $\mathcal{FPC}$, a conceptual formalism where both taxonomical and partonomical relationships can be modeled and complex constraints among components and sub-components can be described by means of a constraint language.

Two inference mechanisms have been defined: *hypothesis formulation* for hypothesizing the set of complex products a set of components can be part of. *Hypothesis validation* verifies whether it is possible to build an instance of a complex product which includes all the components chosen by the customer by taking into account the taxonomy, the partonomy, the constraints among components as well as the user requirements. This mechanism is also able to point out which components have to be added in order to properly configuring the product.

The paper discusses how the combination of hypothesis formulation and hypothesis validation can be used for supporting a customer in selecting and configuring a product. The domain of Personal Computers and their related components is used as a test bed of the system.

1 Introduction

In recent years the diffusion of on-line services and in particular the development of business to consumer e-commerce has shown the central role of recommending ([10]). In fact, more and more researchers have recognized the need to support the client in selecting the goods (or services) sold in a virtual store. Most recommending systems (see for example [9]) focus their attention on mechanisms able to relate preferences (directly expressed by the user or inferred by the system) with the characteristics of the products. Such an approach implicitly assumes that the task of recommending can be reduced to a form of classification (where the requirements of the user are matched with the characteristics of the products) followed by a step of ranking the products according to some measure of the degree of match (see for example [1]). This approach works as long as the products can be viewed as entities without a structure in terms of sub-parts and therefore entities can be described by means of a vector of pairs $\langle attribute, value \rangle$ (for example, movies, restaurants, books can, in many cases, be described in such a way). However, there are domains where the task of recommending involves more complex reasoning mechanisms since there is no single product that can satisfy the requirements of the client. The solution cannot be selected among a set of predefined entities, but has to be synthesized by assembling and configuring a set of products in order to form a complex product able to meet the user requirements[1]. In this way the task of recommending can be described as a task of configuration.

In such kinds of applications, the language used for describing the products should be rich enough to capture the complex set of relations existing among the different products and the constraints which specify what kind of products can be assembled starting from sub-components.

In the present paper we focus our attention on the description and configuration of products in the framework of web stores able to customize both the recommendation and the interaction to the needs of a client. The expertise gained in SeTA, a multi-agent architecture for personalized interaction with the client of a virtual store (see http://www.di.unito.it/∼seta for a in-depth description of the project), is relevant for the present project since in SeTA a number of problems related to the interaction with different kinds of users have been solved. However, in SeTA the client can navigate the virtual catalogue and the system is able to suggest specific products that best match his/her needs and preferences (partly provided directly by the client and partly inferred by the system). In such an activity there is no need of combining together sub-components in order to provide the functionality required by the user since there are specific products satisfying more than one functionality.

On the contrary, in the present project we aim at providing the user with suggestions and recommendations in domains where:
- products can be assembled and configured starting from components
- the functions and the performances of a product can be extended or upgraded by adding or substituting one or more components.

The attention to the needs of different kinds of clients (some of them possibly naive of the domain) has an impact not only on the style of interaction (not discussed in the present paper) but also on the representation issue. In fact, the need of making understandable to the user the selection and configuration task requires a description of the products which is able to characterize them in terms of features as well as to express complex relations among products and components. As a test bed of our approach the domain of personal computers has been selected since this is a typical mass product market where configuration is relevant and the need of supporting differ-

[1] This does not only occur in technical domains, but also in financial applications such as portfolio management.

ent kinds of clients with different degrees of expertise and different requirements is of primary importance.

The framework that we present is currently implemented in a prototype that is being developed in Java.

In the present paper, after introducing the basic domain concepts (section 2), we focus our attention on the representation mechanisms adopted for describing the complex products as well as the components and sub-components (section 3), whilst in subsection 3.1 we discuss the constraints language. In section 4 we sketch the kind of services the system has to provide and we briefly present the reasoning mechanisms that, in combination, are able to provide the required services.

2 Basic Domain Concepts

Before describing the main features of the representation language, it is worth pointing out the basic choices in modeling the domain. The most relevant distinction concerns atomic products vs. complex products and such a distinction is based on the need for a given product to take into consideration its parts and sub-components. In particular, **atomic products** are described via features which synthesize the relevant characteristics of the product without referring to their parts while **complex products** include in their description reference to their parts and to the relations among the parts where each part (or component) can be an atomic product or a complex one.

Various kinds of part-whole relations have been studied in [7, 12]. Following the terminology of [7], we are concerned with a part-whole relation of type *complex-component*. It is worth noting that the atomicity of a product is strongly task dependent: for example a keyboard in our test bed domain is modeled as an atomic product because in a virtual store selling personal computers there is little benefit in describing that a keyboard is made up, among others, of keys and that the keys are arranged in a given order on the keyboard and so on. It is obvious that these pieces of information would be needed if the task or the domain of application were different (for example, manufacturing).

Both atomic and complex products have to be described by means of properties and features characterizing them. For example, a keyboard can be described by its make, model, price, language, etc. This is not peculiar just of the atomic objects, also complex object are partially described by features such as make, cost, etc. However, the main characterization of a complex object is obtained via relations with its parts and the constraints that have to be satisfied by the parts. For example, a PC has as part one (or more) hard disk, a CPU, etc. If a PC has a hard disk of type SCSI and a motherboard of type EIDE, then a controller SCSI is needed. These constraints have to be explicitly represented in description of complex products.

Because of the importance of the relation between a (complex) product and its parts, this relation has a special status and is modeled via the introduction of **"has-part" relationship**. For our purpose it is sufficient to introduce only one kind of *has-part* relation (the one between each complex and its components) and we can consider it as an antisymmetric and transitive relation (for the issues raised by the transitivity of *part-whole* relations see also [12, 7, 3]). *has-part* is therefore the basic mechanism for defining the *partonomy* between the products described in the domain. However, also *taxonomic relations* have to be defined in order to allow the definition of more specific or more general concepts. For example, in our domain we want to describe that there are different kinds of printers such as ink jet printers and laser printers. This requires the introduction of relations of superclass and subclass for capturing the taxonomic relation and consequently of appropriate mechanisms of inheritance.

3 The Conceptual Language $\mathcal{FPC}$

We present here a language to describe the *classes* (both of atomic products and of complex products) and the relationships among them. As concerns the *individuals* (i.e. the instances of the classes), we shall briefly describe their representation in section 4.

The conceptual language $\mathcal{FPC}$ (Frames, Parts and Constraints) is, on one hand, a simple frame-based language able to represent taxonomic relations among classes. On the other hand, $\mathcal{FPC}$ offers the facility for building *partonomies* to describe the structure of complex objects in terms of their parts. Moreover, $\mathcal{FPC}$ is enriched with a powerful language for expressing complex constraints among the components and the sub-components of complex objects.

Each frame represents a class of objects and it is characterized by a type: *atomic* or *complex* for the classes of atomic or complex products (respectively). Besides the type, each frame F contains the following information:
– its **name**: the name of the class that it describes;
– the slot **superclass** contains the name of the frame that represents the direct superclass (if any). It is worth noting that the superclass is unique since we do not allow multiple inheritance for sake of simplicity.
– the slot **subclasses** contains the names of the frames which represent the direct subclasses (if any) of the class[2]. We assume that if a class has more than one direct subclasses, these are two by two disjunct. Moreover the slot **subclasses** has the facet *partitioned* whose boolean value denotes whether the subclasses mentioned in the **subclasses** slot form a partition of the class.

The slots **superclass** and **subclasses** express the subsumption relationship among classes: a class is subsumed by its superclasses (both direct and indirect) and it subsumes all its subclasses (both direct and indirect). A class is trivially subsumed by itself.

The description of the class is made possible by a set of slots which are subdivided into **own slots** and **member slots**, following [5]. While the **own slots** are used for modeling the attributes of the class as a whole (as the "average price" of an hard disk), the **member slots** describe the attributes of the objects belonging to the class (e.g. the make of each CPU in the CPU class) and are *inherited* by any subclass from the superclass. Each slot has a *name*; member slots are characterized by a type: *descriptive* or *partonomic*. Slots of the latter type are used to represent *has-part* relations, whilst the ones of the former type are used to represent all the other kinds of object attributes. The *atomic* frames cannot have any partonomic slot.

Besides the type and the name, each member slot has the facets *CardinalityMin*, *CardinalityMax* (that take values in the set of non negative integers) and *ValueClass* that can take as value a class described by a frame or a subset of a pre-defined set. It is worth pointing out that the formalism imposes a strict inheritance since no exception is admitted following the tradition of KL-ONE like representation formalism ([4]). However, the formalism provides a mechanism for restricting slots along the taxonomy by putting stricter constraints on *CardinalityMin*, *CardinalityMax* and/or *ValueClass*.

For each descriptive member slot, $ValueClass$ can be assigned to a subclass of a pre-defined class (such as integers, reals, strings or booleans) or to a frame.

[2] To be precise, we should always distinguish between a *frame*, which is a language element, and the *class* that it describes. However, in order to simplify the presentation, we will use the words *frame* and *class* as synonyms.

For each partonomic member slot, *ValueClass* must be assigned to a frame (either *atomic* or *complex*). Thus, any partonomic member slot can be viewed as a link between two frames[3]. Moreover, for the sake of simplicity, any complex class (whose direct superclass is different from the root of the taxonomy) can't introduce any new partonomic slot w.r.t. the set of those that it inherits from its superclass. Given a knowledge base KB and a complex class C_1 in KB, we call **slot chain (starting in C_1)** any sequence $\gamma = \langle p_1, p_2, \ldots, p_n \rangle$ $(n \geq 1)$ of member slots such that in KB there are the set $\{C_2, \ldots, C_{n-1}\}$ of complex classes and the class C_n (either atomic or complex) such that C_i $(1 \leq i \leq n-1)$ has a partonomic member slot named p_i (possibly inherited) whose $ValueClass$ is C_{i+1} and C_n has a member slot (either descriptive or partonomic and possibly inherited) named p_n whose $ValueClass$ is a set D. D is called the *codomain* of the chain γ starting in C_1 and we indicate it with $cod(C_1, \gamma)$.

Because of their importance for the constraint language and for the inference mechanisms, we give a precise semantic for the member slots and for the sets of slot chains.

Given a class C, each member slot s of C represents a relation from C to the class D assigned to the facet $ValueClass$ of s. If the slot s has $CardinalityMin = m$ and $CardinalityMax = n$, then $(\forall a \in C)(s(a) \subseteq D \wedge m \leq |s(a)| \leq n)$[4]. In this case, we write $C_{min}(C, \langle s \rangle) = m$ and $C_{max}(C, \langle s \rangle) = n$. We say that the pair $\langle m, n \rangle$ and the set D are, respectively, the *cardinality restriction* and the *value restriction* for (or the codomain of) the slot s.

The semantic of a member slot is naturally generalized to any slot chain: a slot chain $\gamma = \langle p_1, p_2, \ldots, p_n \rangle$ $(n \geq 1)$ starting in C_1 and such that $cod(C_1, \gamma) = D$ represents the relation composition $p_n \circ p_{n-1} \circ \ldots \circ p_1$ from C_1 to D. Moreover, the set $R = \{\gamma_1, \ldots \gamma_m\} (m \geq 1)$ of slot chains (each starting in C_1) represents the relation union $\bigcup_{i=1}^{m} \gamma_i$ from C_1 to $cod(C_1, R) = \bigcup_{i=1}^{m} cod(C_1, \gamma_i)$.

Let C be a complex class and p one of its partonomic slots. If $C_{min}(C, \langle p \rangle) = m$, $C_{max}(C, \langle p \rangle) = n$ and $cod(C, \langle p \rangle) = D$, it means that any complex object belonging to the class C has a set of components belonging to the class D in a number that ranges from m to n. Such components could be atomic objects or complex objects: in the former case D is an atomic class, while in the latter D is a complex class.

Exclusivity Assumptions on Parts: For each pair of complex classes C_1 and C_2 (possibly $C_1 \equiv C_2$), the following hold:

– if p_1 and p_2 are two *different* partonomic slots of C_1, then $(\forall c_1 \in C_1)(p_1(c_1) \cap p_2(c_1) = \emptyset)$;

– if p_1 and p_2 are two partonomical slots of C_1 and C_2 respectively (possibly $p_1 \equiv p_2, if C_1 \equiv C_2$), then $(\forall c_1 \in C_1)(\forall c_2 \in C_2)(c_1 \not\equiv c_2 \rightarrow p_1(c_1) \cap p_2(c_2) = \emptyset)$.

It should be clear that one important consequence of the preceding assumption is that if two different instances of complex objects share a same component, then one of them contains the other. Moreover, on the basis of this assumption, we can generalize the cardinality restriction concept both to the slot chains and to the sets of slot chains in the following way (γ and R are defined above):

– $C_{min}(C_1, \gamma) = \prod_{i=1}^{n} C_{min}(C_i, \langle p_i \rangle)$;

– $C_{min}(C_1, R) = \sum_{i=1}^{m} C_{min}(C_1, \gamma_i)$.

The case of the maximum cardinalities is analogous.

Summarizing, each knowledge base consists of (at least) two different taxonomies: one for the classes of complex products and another for the classes of atomic products; each taxonomy is a tree because we restrict to single inheritance. A distinctive feature of the $\mathcal{FPC}$ concerns the treatment of has-part relation by means of the introduction of partonomic slots.

3.1 Constraint Language

Each frame modeling a complex class C contains a set (possibly empty) of constraints that specify those restrictions on the components and the sub-components of the objects in C that couldn't be expressed using only the cardinality and the value restrictions for the member slots.

Let C be a complex class of a knowledge base KB and CC its set of constraints. Any constraint in CC is of the form $\alpha \Rightarrow \beta$, where α is a conjunction of predicates or the boolean constant **true** and β is a predicate or the boolean constant **false**. The meaning is that for every object c in C, if c satisfies α then it must satisfy β. This must be true for each constraint in CC. It should be clear that if $\alpha = \mathbf{true}$, then, for each object in C, β must always hold, while if $\beta = \mathbf{false}$, then, for each object in C, α can never hold. The constraint $\mathbf{true} \Rightarrow \mathbf{false}$ is meaningless.

Due to the space limitations, we describe here a simplified version of the constraint language that we actually defined and implemented.

The basic building blocks of the constraint language are the predicates. Let $R = \{\gamma_1, \ldots, \gamma_m\} (m \geq 1)$, where each $\gamma_i = \langle p_{i_1}, \ldots p_{i_n} \rangle (i_n \geq 1)$ is a slot chain starting in C. We indicate with R both the set of slot chains and the relation that it represents. We have six kinds of predicates for C[5]:

1) $(R)(h, k)$.

$c \in C$ satisfies the predicate *iff* $h \leq |R(c)| \leq k$, where h, k are non negative integers with $h \leq k$.

2) $(R)(inI)$. $I = I_1 \cup \ldots \cup I_s (s \geq 1)$ and each $I_i (i = 1, \ldots s)$ is a class.

$c \in C$ satisfies the predicate *iff* $R(c) \subseteq I$.

3) $(R)((inI^1(s^1, t^1)) \ldots (inI^{(r)}(s^{(r)}, t^{(r)})))$. $r \geq 1$. $I^{(i)} = I_1^{(i)} \cup \ldots I_s^{(i)}$ $(i = 1, \ldots, r)$ and each $I_j^{(i)}$ $(j = 1, \ldots, s)$ is a class.

$c \in C$ satisfies the predicate *iff* $\bigwedge_{i=1}^{r} (s^{(i)} \leq |R(c) \cap I^{(i)}| \leq t^{(i)})$.

4) $(\Sigma(R))$ $RelOp$ n. $RelOp \in \{=, <, >, \leq, \geq\}$. n is a number and $cod(C, R)$ is a numeric set.

$c \in C$ satisfies the predicate *iff* $(\sum_{i \in R(c)} i)$ $RelOp$ n.

5) $(\Sigma(R))$ $RelOp$ $(\Sigma(S))$. S is a set of slot chains starting in C $(S \neq R)$. It is analogous to the predicate 4.

6) $(\gamma_i) = (\gamma_j)$. γ_i and γ_j $(\gamma_i \neq \gamma_j)$ are two role chains starting in C. $cod(C, \gamma_i)$ and $cod(C, \gamma_j)$ must be of the same kind.

$c \in C$ satisfies the predicate *iff* $\gamma_i(c) = \gamma_j(c)$, where $\gamma_i(c)$ and $\gamma_j(c)$ are, in general, two multisets.

Let $cc = \alpha \Rightarrow \beta$ and ψ be a constraint and a predicate for the complex class C in a knowledge base KB, respectively.

[3] No cycle (possibly involving also "subclass links") is allowed.

[4] As in the case of frames, we should distinguish from the *slot*, which is a language element, and the *relation* that it represents. However, to simplify the presentation, we use the slot name also to indicate the relation represented by the slot. The simplification adopted for slot chains is analogous.

[5] In the predicates, $R(c)$, $S(c)$ and the various $\gamma_i(c)$ can be, in general, multisets. We briefly give some definitions that generalize the equality and the inclusion relations as well as the cardinality and the intersection operators to multisets. If S is a multiset, any element a occurs in S $times(a, S)$ times. Let $\overline{S} = \{a / \ times(a, S) > 0\}$ ($\overline{S}$ is a **set**); we have (if T is a multiset): $S = T$ iff $(\forall a)(times(a, S) = time(a, T))$; $S \subseteq T$ iff $\overline{S} \subseteq \overline{T}$; $|S| = \sum_{a \in \overline{S}} times(a, S)$; $S \cap T = U$, such that $\overline{S} \cap \overline{T} = \overline{U} \wedge (\forall a \in \overline{U})(times(a, U) = max\{times(a, S), times(a, T)\})$.

If the taxonomic and the partonomic description of C entails cc (ψ), cc (ψ) is said $\mathbf{C_{TP} - tautological\ in\ KB}$.

If cc (ψ) is consistent with the taxonomic and the partonomic description of C, cc (ψ) is said $\mathbf{C_{TP} - consistent\ in\ KB}$ (otherwise, it is said $\mathbf{C_{TP} - inconsistent\ in\ KB}$).

Inference mechanisms have been developed that are able to single out the $C_{TP} - tautological$ predicates/constraints and the $C_{TP} - inconsistent$ predicates/constraints. Such mechanisms are sound but not complete.

In each knowledge base, the constraints CC associated to a class C are consistent both among them and with the taxonomic and partonomic description of C as well as with all the constraints associated to the classes involved in this description. Moreover, CC doesn't contain any constraint that the system would recognize as $C_{TP} - tautological$.

Due to the novelty of the **constraints**, we describe the rules for their **inheritance** from a class to a subclass: let $C^{'}$ be a subclass of C.
– If cc is recognized as $C^{'}_{TP} - tautological$, cc is not (explicitly) inherited by $C^{'}$;
– otherwise, $C^{'}$ inherits cc in the form $\alpha^{'} \Rightarrow \beta^{'}$, where $\alpha^{'}$ is obtained by deleting from α all the predicates recognized as $C^{'}_{TP} - tautological$. If, after this deletion, no predicate remains, $\alpha^{'}$ is the constant **true**. If β is recognized as $C^{'}_{TP} - inconsistent$, $\beta^{'}$ is the constant **false**, otherwise, $\beta = \beta^{'}$.

It should be clear that the previous rules provide a way to reduce the redundancy in the constraint inheritance and to avoid some useless controls to the inference mechanisms.

4 Inference Services

The inference mechanisms defined on the $\mathcal{FPC}$ language have to take into account the role of a recommending system in a virtual store where products have to be configured. It is important to note that in such kind of applications there is a wide variety of customers with very different degrees of domain expertise. The system has to support the naive customer by checking his/her choices of products and by suggesting products which meet his/her requirements. However, in some cases the customer is an expert and he/she cannot be required to answer too many questions of the system [6].

Depending on the stage of the interaction and the kind of user, the system has to perform inferences on the basis of different inputs from the user:
– **scenario 1:** the customer is selecting products and he/she puts them in the virtual cart. The system in order to be able to check if the products can work together, has to figure out what kind of complex product the user has in mind. The system has to use the partonomic and the taxonomic description for hypothesizing the goal of the customer. This kind of abductive inference is useful for focussing the interaction with the user, for example for asking him/her which of a set of complex products he/she is interested in.
– **scenario 2:** the customer has made explicit the complex product she/he is interested in and has selected many (possibly all) components for such a product. The system has to verify whether the constraints associated with the classes in the knowledge base are satisfied by the current set of choices of the user and possibly suggest the insertion of some missing components.

[6] In this paper we do not address the problem of adapting the interaction to the different kinds of users since many of these problems have received a satisfactory solution within the SeTA system ([2]).

– **scenario 3:** the customer is able just to provide some requirements on the product (e.g. the maximum price for a PC, ...), and possibly to indicate some specific components to be included in the product (e.g. a particular CPU, ...). In such a situation the task of the system is similar to the previous one, but an extra inference step is necessary since the system has to know which kind of constraints a complex product should match in order to provide the facilities and performances required by the user.

Usually, the customer of a virtual store browses an electronic catalogue containing the description of the products sold in the store. In our case, the electronic catalogue contains both the descriptions of the various kinds of complex products and those of the atomic ones. Each atomic product appearing in this catalogue is associated to a leaf of the atomic products taxonomy. This leaf provides the most specific description of the product needed by the inference mechanisms. The atomic products chosen by the customer are grouped by their most specific class and are represented by the set $\mathcal{CH} = \{ch_1, \ldots, ch_n\}$ $(n \geq 1)$. With $msc(ch_i)$ and $mult(ch_i)$ $(i = 1, \ldots, n)$ we denote, respectively, the most specific class to which the atomic product ch_i belongs (i.e. the leaf atomic class of the taxonomy associated to it) and the number of such kind of products chosen by the customer. In $\mathcal{CH}$ there can't be any repeated class, i.e.: $(\forall ch_i, ch_j \in \mathcal{CH})(i \neq j \rightarrow msc(ch_i) \neq msc(ch_j))$.

To describe the inference mechanisms, we need the following notation. Let C_1 and C_2 be two classes of a knowledge base. If $\gamma = \langle p_1, \ldots, p_n \rangle$ $(n \geq 1)$ is a slot chain such that:
– γ starts in C_1;
– $cod(C_1, \gamma)$ subsumes C_2;
– $(\forall 1 \leq k \leq n-1)(\exists\, a\, class\ D)$
$(cod(C_1, \langle p_1, \ldots, p_k \rangle)\ subsumes\ D \quad \wedge \quad \neg partitioned(D) \quad \wedge$
$cod(D, \langle p_{k+1}, \ldots, p_n \rangle)\ subsumes\ C_2)$,
we write $C_1\ [\gamma] \triangleright\ C_2$.

We write also $C_1 \triangleright C_2$ to indicate that $(\exists \gamma)(C_1 [\gamma] \triangleright C_2)$; in this case, we say that C_2 is **more specific than** C_1 **w.r.t. the partonomy** [7]. It is worth noting that if any element of a class C_2 can be a component of any element of a class C_1, then $C_1 \triangleright C_2$.

1. HYPOTHESIS FORMULATION. This mechanism is aimed at hypothesizing which kinds of complex products a customer is interested in, on the basis of the atomic products that he/she has put in his/her virtual shopping cart. In particular, this mechanism individuates, among all the complex classes, those whose elements can't have as parts the kinds of atomic products chosen by the customer. All the other classes of products are considered as *plausible hypotheses* and are used either to focus the interaction with the customer and possibly change the set of plausible hypotheses (**scenario 1**) or to perform a set of hypothesis validation steps by means of the mechanism 2 (**scenario 2**).

The algorithm can be sketched as follows:

INPUT: a knowledge base KB and $\mathcal{CH} = \{ch_1, \ldots, ch_n\}$ $(n \geq 1)$
OUTPUT: HYP (the set of plausible hypotheses).
$HYP := \{C / C\ is\ a\ complex\ class\ in\ KB \wedge \neg partitioned(C) \wedge (\forall ch \in \mathcal{CH})(C \triangleright msc(ch))\}$;
Rank the hypotheses in HYP according to the $\triangleright$ relation;
return HYP.

As can be seen, in HYP there are only classes whose direct subclasses are not a partition of them. This means that the hy-

[7] If C is a class, $partitioned(C)$ *iff* the direct subclasses of C (if any) are a partition of C.

potheses in HYP are as specific as possible w.r.t. the taxonomy. Moreover, it should be clear that if $HYP = \emptyset$, then there doesn't exist any complex product whose components (and sub-components) are of the kinds specified in $\mathcal{CH}$.

2. HYPOTHESIS VALIDATION. Given a complex class C [8] and a set $CONSTR_I$ of input constraints for C (expressed in the language defined in section 3.1, but different from those associated to the class C in the knowledge base), the algorithm tries to build an instance c of the class C that satisfies all the constraints $CONSTR_I$. It is worth noting that such an instance can be built iff the set of constraints $CONSTR_I$ is consistent with the whole description of C in the knowledge base (i.e. with both the taxonomic and the partonomic description of C and with all the constraints associated to the complex classes involved in the definition of C). The main idea is straightforward: at the beginning there is only the hypothesis that the component c exists. Then, the algorithm tries to progressively assemble a set of components and sub-components in order to build c. During this process, it takes into account both the description of the class C and the set $CONSTR_I$. If this process succeeds in building the instance, then it returns it, otherwise it signals that the set of input constraints can't be satisfied (and this proves the initial hypothesis wrong).

An *instance of a complex product* $c \in C$ can be represented as a graph with a root representing c and in which the nodes represent the components and the sub-components of c. Each arc is labelled with a partonomic slot name. If two nodes m and n (representing, say, $d_1 \in D_1$ and $d_2 \in D_2$, respectively) are connected by a path from m to n labelled $\langle p_1, \ldots, p_n \rangle$ ($n \geq 1$), it means that $d_2 \in \gamma(d_1)$, where $\gamma = \langle p_1, \ldots, p_n \rangle$ is a slot chain and $D_1\ [\gamma] \triangleright\ D_2$.

Thanks to the exclusivity assumption on parts (see section 3), any instance graph is a tree (*the instance tree*).

The hypothesis validation algorithm starts with a tree containing only the root representing the hypothesis c and it progressively expands the instance tree for c. This progressive expansion is actually a search process in which, usually, some choices are performed. We adopt a chronological backtracking mechanism to change the past choices when the tree under construction can't grow any more without violating any constraint.

Rather informally, we can say that the algorithm always work with a **hypothetical instance** of a class C, i.e. with an instance tree, possibly not completely specified, that doesn't violate neither the description of C, nor the input constraints $CONSTR_I$. However, such a tree can actually provide only an incomplete description of the instance. Therefore, it is possible that the truth value of some predicates (and thus of some constraints) can't be determined for a *hypothetical instance*. Thus, it is important to stress that saying that a *hypothetical instance* doesn't violate any constraint (either contained in the knowledge base or in $CONSTR_I$) is not the same thing as saying that it *satisfies* all the constraints nor as saying that its tree can be expanded in some way such that every constraint will eventually be satisfied! A *hypothetical instance* whose tree *can be* completely expanded in some way that satisfies all the constraints is said an **instance**. Any *instance* whose tree is completely expanded is said a **completely described instance**, otherwise it is said a **partially described instance**.

It is clear that if $CONSTR_I$ can't be satisfied, the algorithm ends after having performed all the possible choices. If the hypothetical instance c is really an instance, the algorithm can't stop before having discovered this fact. However, the number of the furhter instance tree T expansion steps is task dependent. If the goal is to provide a support for a complete configuration of the product, the expansion can continue until a **completely described instance** is produced. Otherwise, if the main goal is to test the consistency of a set of constraints (as in the **scenarios 2 and 3**) given by the customer, a lazier approach that can return a **partially described instance** is more suitable. We sketch here the algorithm in its lazy version currently implemented in the prototype. The main idea is that the expansion of the instance tree is performed by considering only those slots whose cardinality and whose fillers' classes can be critical for the satisfaction of the input constraints $CONSTR_I$. To do so, the sets C_c and P_c are associated and maintained for each component. The former initially contains all the constraints *bound* to the input ones[9]. After an expansion step, C_c is updated in order to remove from it the constraints that became satisfied. The set P_c is computed on the basis of C_c and it contains all the slots that can influence the truth values of the constraints in C_c. A constraint propagation mechanism is used to reduce the number of the alternatives produced by the functions computeAdmissibleCardinalities(p) and computeAdmissibleClasses(m).

Let T be the instance tree for $c \in C$, containing only the root (in the following it is intended that the BACKTRACKING call returns the *failure* message if no alternative choice is possible).

INPUT: a knowledge base KB, T, $CONSTR_I$.
OUTPUT: an instance tree for the *instance* $c \in C$ or the *failure* message.
if $(\exists cc \in CONSTR_I)(cc\, is\, recognized\, as\, C_{TP} - inconsistent)$
  then return FAILURE;
Let $CONSTR_I^{'}$ be $CONSTR_I$ from which the constraints recognized as $C_{TP} - tautological$ have been removed;
if $CONSTR_I^{'} = \emptyset$ **then return** T;
while (< there is in T some not marked components (i.e. nodes)>)
do begin
n := chooseComponent(T); /*Currently handled with a queue*/
C_c := computeCurrentConstraints(n);
P_c := computeCurrentSlots(n,C_c);
p := chooseSlot(P_c); /*Currently random*/
$cards$:= computeAdmissibleCardinalities(p);
if $cards = \emptyset$ **then** BACKTRACKING;
$card(p)$:= chooseCardinality($cards$); /*Currently the minimum one*/
if $|cards| > 1$ **then**
  < save $cards - \{card(p)\}$ as alternative choices >;
if < T violates some constraint in C_c > **then** BACKTRACKING;
< Expand T by introducing $card(p)$ new nodes, each one connected with n via an arc labelled p >;
for each < new created node m > **do begin**
  $classes$:= computeAdmissibleClasses(m);
  if $classes = \emptyset$ **then** BACKTRACKING;
  $class(m)$:= chooseClass($classes$); /*Currently random*/
  if $|classes| > 1$ **then**
    < save $classes - \{class(m)\}$ as alternative choices>
  end
C_c:= updateCurrentConstraints(n);
if < T violates some constraint in C_c > **then** BACKTRACKING;
P_c := updateCurrentSlots(n);
if $P_c = \emptyset$ **then** < mark the component n >
end

[8] For the sake of simplicity, we assume $\neg partitioned(C)$.

[9] The current system relies on a formal definition of a bind relation among constraints.

return T.

Translation of the Set $\mathcal{CH}$ in a Set of Constraints. As concerns the **scenario 2** depicted at the beginning of this section, the task of controlling that the set of components chosen by the customer (and represented by the set $\mathcal{CH}$) can actually be parts of a complex product of kind C (i.e. belonging to the class C [10]) can be performed by the hypothesis validation algorithm. Therefore, there is the need to translate the set $\mathcal{CH}$ in a set $CONSTR_I$ of input constraints for the class C. This can be done in the following way:
$CONSTR_I := \emptyset$;
for each $ch \in \mathcal{CH}$ **do begin**
$R(ch) := \{\gamma / C\,[\gamma] \triangleright\ msc(ch)\}$;
$CONSTR_I := CONSTR_I \cup \{\mathbf{true} \Rightarrow (R(ch))((in\ msc(ch)(mult(ch), C_{max}(C, R(ch)))))\}$
end.
It should be clear (see the description of the predicate 3 in section 3.1) that the algorithm 2, with the class C and this set $CONSTR_I$ as inputs, verifies if it is possible to build an instance $c \in C$ containing at least $mult(ch)$ elements of kind $msc(ch)$, for each $ch \in \mathcal{CH}$ chosen by the customer. Moreover, if such an instance is possible, all the components that were not chosen by the customer, but that have been added during the expansion of the instance tree, can be the basis for suggesting some needed components to the customer.

We just point out that the class C could have been indicated by the customer autonomously or after a dialogue with the system on the basis of the results of the hypothesis formulation algorithm.

5 Conclusions

In the present paper we have described the main features of $\mathcal{FPC}$, in particular the role played by has-part relations and constraints in modeling complex product. We have also sketched some basic reasoning mechanisms which can be exploited to support a customer in selecting and configuring a product. The adoption of a declarative approach to knowledge representation has the advantage of making a clear distinction between the formalism and the reasoning mechanisms. $\mathcal{FPC}$ is powerful enough for modeling different types of knowledge, in particular the relations between a complex product and its components as well as the descriptive characteristics of the products. This ability is very relevant since in a business to consumer perspective of e-commerce a recommending system has also the task of making intelligible to the customer the description of the product. For this reason the products cannot be described just by means of technical features, but a description should include also economic, functional, aesthetic features. The recommending system has to use these pieces of information for selecting and ranking products to be suggested to the user, but can also use these pieces of information for customizing the description of the product by taking into consideration the user profile (see, for example, [2]). In the paper we have focussed our attention to the needs of the customer, but we also have started to consider the services useful to the manager of the virtual store. For example, we have developed some additional inference mechanisms able to single out some type of inconsistency in the knowledge base. These mechanisms are quite useful to knowledge engineers when the knowledge base has to be updated and/or extended or a new domain has to be modeled.

It is worth noting that even if the representation language and the inference services that we described are tailored to the representation of (and to the reasoning on) products in a virtual store, they have a few similarities with some mechanisms used in the technical configuration task. In [11] the Generative Constraint Satisfaction Problem (GCSP) approach is presented and in [6] a configurator for large telephone switching systems based on the GCSP paradigm is described. Although in the framework that we proposed the configuration (or the testing of the user requirements consistency) task is not defined as a (G)CSP, the *hypothesis validation* algorithm does actually work over a set of variables representing slots and components whose number can't be predetermined. The activation of the slot variables is dynamically performed by the function computeCurrentSlots(n,C_c) and it is analogous to the activation of the property variables in GCSP. However, in our framework, the *activation constraints* are implicitly contained in the partonomic description of the classes[11]. The activation of new components variables is based on the value chosen for (the cardinality of) a (current) slot and it represents the instance tree expansion step. As the problem solver described in [11, 6] for the GCSP, the *hypothesis validation algorithm* is based on a backtracking architecture in which some constraint propagation mechanisms are used to reduce the set of possible choices (i.e. to reduce the domain of the slot and of the component variables).

Moreover, the semantic of frames and that of member slots (chains) follow that of description logic (DL) terms (a DL-based configurator is described, for example, in [8]). However, differently from most DL-based systems, our system doesn't include a classificator and the taxonomies have to be explicitly built by the knowledge engineer.

REFERENCES

[1] M. Albers, C. M. Jonker, M. Karami, and J. Treur, 'An electronic market place: Generic agent models, ontologies and knowledge', in *Agent-Based Decision-Support for Managing the Internet-Enabled Supply-Chain*, (1999).

[2] L. Ardissono, A. Goy, G. Petrone, and M. Segnan, 'Adaptive user interfaces for on-line shopping', in *Proc. of the Adaptive User Interfaces Spring Symposium of AAAI*, (2000).

[3] A. Artale, E. Franconi, N. Guarino, and L. Pazzi, 'Part-whole relations in object-centered systems: An overview', *Data and Knowledge Engineering*, (20), 347–383, (1996).

[4] R. J. Brachman and J. G. Schmolze, 'An overview of the kl-one knowledge representation system', *Cognitive Science*, (9), 171–216, (1985).

[5] R. Fikes and T. Kehler, 'The role of frame-based representation in reasoning', *Comm. ACM*, **28**(9), 904–920, (1985).

[6] G. Fleischanderl, G. E. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner, 'Configuring large systems using generative constraint satisfaction', *IEEE Intelligent Systems*, (July/August 1998), 59–68, (1998).

[7] P. Gerstl and S. Pribbenow, 'Midwinters, end games, and body parts: a classification of part-whole relations', *Int. J. Human-Computer Studies*, (43), 865–889, (1995).

[8] D. L. McGuinness and J. R. Wright, 'An industrial-strength description logic-based configurator platform', *IEEE Intelligent Systems*, (July/August 1998), 69–77, (1998).

[9] B. Raskutti, A. Beitz, and B. Ward, 'A feature-based approach to recommending selections based on past preferences', *User Modeling and User-Adapted Interaction*, (7), 179–218, (1997).

[10] P. Resnick and H. Varian (Eds.), 'Recommender systems', *Comm. ACM*, **40**(3), 56–89, (1997).

[11] M. Stumptner and A. Haselböck, 'A generative constraint formalism for configuration problems', in *LNAI 728*, pp. 302–313, (1993).

[12] M. E. Winston, R. Chaffin, and D. Herrmann, 'A taxonomy of part-whole relations', *Cognitive Science*, (11), 417–444, (1987).

[10] Again, we assume $\neg partitioned(C)$.

[11] Due to the *lazy policy* of the algorithm, not all the partonomic slots of a class are introduced for each component belonging to that class and occurring in an instance tree.

Configurable Software Product Families

Tomi Männistö[1], **Timo Soininen**[1] and **Reijo Sulonen**[1]

Abstract. Product configuration is a specific area of research and business for mechanical (and electrical) products. However, configurable software products have not attracted as much interest. This paper outlines the concept of configurable software product families covering millions of variants from which product individuals are configured to meet particular customer needs. Solutions to managing such software products are sought from experiences with mechanical products and expressed here in the form of a research agenda.

1 INTRODUCTION

This paper investigates software as a configurable product family that may include millions of variants. Thus, we aim at providing methods and tools for a software engineering paradigm in which instances of software are created in a routine manner. This creation would be based on a predefined model (or architecture) that describes all variants and the required knowledge for selecting functional combinations of components. Such paradigm becomes important, for example, when software is embedded in a product that is configurable and the software must adapt to the hardware configuration. If the available memory is limited, the loaded software cannot simply include all possible variability and dynamically adapt to the hardware. Examples of such products are tomorrow's mobile terminals. Similar strategies are also currently sought for enterprise resource planning (ERP) systems, which are large configurable information system packages [1].

In this paper, we chart a research agenda starting from the experiences in product configuration of mechanical and electrical products, which are called *traditional products* in the following. We begin with a brief introduction to product configuration and software engineering, especially software architectures. However, we assume that the reader is familiar with product configuration and thus the introduction to it is minimal.

Our goal for the research work is to find description methods for software product families. The methods should be understandable to software engineers with no special skills in formal methods or logic. In addition, the description of a software product family of an industrial company should be manageable both in complexity and in size. This restricts the modeling essentially to the design level, as very deep models tend to be extremely large. Furthermore, the used description language should allow the models to be processed by computers, which requires strict tradeoffs between the expressivity and complexity of the underlying concepts, as the more powerful logical formalisms may in practice be infeasible.

With respect to previous work in software engineering, a natural counterpart to which this research should be contrasted is modeling and applying software architectures. We aim at rather limited models if compared to the most general approaches in software engineering but, on the other hand, we aim at general concepts that are not specific to any particular software domain.

[1] Helsinki University of Technology, TAI Research Centre and Laboratory of Information Processing Science, P.O. Box 9555, FIN-02015 HUT, Finland. Email: {Tomi.Mannisto, Timo.Soininen, Reijo.Sulonen}@hut.fi

2 PRODUCT CONFIGURATION

A *configurable product* is adapted to the needs of a particular customer in a *configuration process* using predesigned *components* and a predefined *configuration model*. A *configuration task* is thus to find a suitable variant from the search space defined by the configuration model. The output of a *configuration process* is a *configuration,* which is an adequate description of the *product individual* so that it can be manufactured (see Figure 1) [2,3,4,5,6,7].

3 SOFTWARE ENGINEERING

In this section, a brief look is taken at the basic concepts in software engineering, in particular those of software architectures. We begin briefly with component based software, move on to software architectures and concentrate there on software architecture description languages and domain specific software architectures, which provide the central point of reference for this paper.

Within component-based software engineering (CBSE), definitions of a *software component* include [8]:

- nearly independent and replaceable part of a system with a clear function in the context of a well-defined architecture
- dynamically bindable package that is accessed through documented interfaces
- unit of composition with contractually specified interfaces
- business component representing an autonomous business concept or process.

In many cases, components are seen as rather independent units and it is required that truly composable systems allow connecting system components into a whole in ways not foreseen by the original developers of the components [9].

Software architecture is a term understood in many different ways, typically meaning the structure of components of a software system, including the relationships and guidelines for design and management of evolution [10]. According to Moriconi et al. [11], a software architecture is represented by the following concepts:

- component
- *interface* that denotes a logical point of interaction between a component and its environment
- *connector,* relating interface points, components or both
- *configuration,* which is a collection of constraints that wire objects into a specific architecture
- *mapping* from the language of an abstract architecture to the language of concrete architecture
- *architectural style*

Architectural style is defined by a collection of conventions for a class of software architectures. Style is thus more a general theory for a subfield of software engineering. Common architectural styles

include pipe-filter, batch-sequential, blackboard, implicit invocation (event-based) and client-server [11]. Formal methods allow analyzing properties of styles and result to a set of general theorems about all systems in the family [12].

Software architecture description languages (ADL) are used to support architecture-based system development. *A system architecture* or *architectural model* is specified by a set of components, connectors, a configuration and a set of constraints and is written in an architecture description language [13]. An architectural model may apply to a single system or to a family of systems in a domain; the latter is referred to as a *generic architecture* or *domain specific software architecture* (DSSA*)* [14], which comprises of [15]:

- *reference architecture* describing a computational framework for a domain of applications
- *component library* of reusable chunks of domain expertise
- *application configuration method* for configuring components to meet particular application requirements

The focus of this paper is on software product families that include large variation. Variety in software product families can be implemented by an approach called *customization,* in which a 'universal' software product is adapted to behave as any specific variant [16]. However, the size of the software product increases because it contains all the variability of the product.

An alternative approach is to use preprocessor directives to optionally include pieces of source code. In this approach, however, the big picture is easily lost, as the representation of variation is not explicit but is embedded in the source code. Variability may also be achieved by *modularization,* in which the variants are produced by selecting appropriate components to the family architecture [15,16].

When large software products are adapted to different customers, a typical approach is "copy and paste", also called *cloning*. That is, an existing variant of the software product is taken as a basis and then modified accordingly. This approach has its drawbacks since in duplication and ad doc modification of architectural components the original ideas behind the architecture are easily lost, which in consequence deteriorates the overall product architecture [17].

4 CONFIGURING SOFTWARE

4.1 Feasibility of Configuring Software

Some issues specific to software might make configurable software products infeasible. For example, including extraneous components does not typically increase the cost of a product individual. This enables in many cases the selling of software product individuals that contain all possible features. Sometimes, however, the available memory is a limiting resource. In such case, it may be necessary to carefully select the components in a particular configuration simply because otherwise the product individuals would not fit into the memory.

In mobile communication, there are also limitations in bandwidth, which may become an important factor if the software product individual is transferred via a wireless communication channel. Therefore, it may be important to load the exact software variant for the hardware in question and take into account the hardware and other software options already installed into the product individual to avoid unnecessary usage of bandwidth.

The current generation of ERP systems relies on a monolithic software architecture in which customer requirements are met by a large number of parameters, options and configurable functionality. However, a minimalist strategy based on components is an alternate way to meet situation-specific requirements [1].

The common point in all these cases is that customer specific variation of software is needed but it should be implemented by other means than single monolithic software product.

4.2 Comparison to Mechanical Products

There are three major product processes: development, order-delivery and after-sales. The processes and their results are illustrated in Figure 1. For configurable products the order delivery process is required to be smooth and not to include any design work, which is carried out in the development process. Many companies manufacturing project-like traditional products have recently sought ways towards this kind of operation, i.e., product configuration. Similarly, software engineering can find ways towards order-delivery process that would provide customer specific solutions without programming.

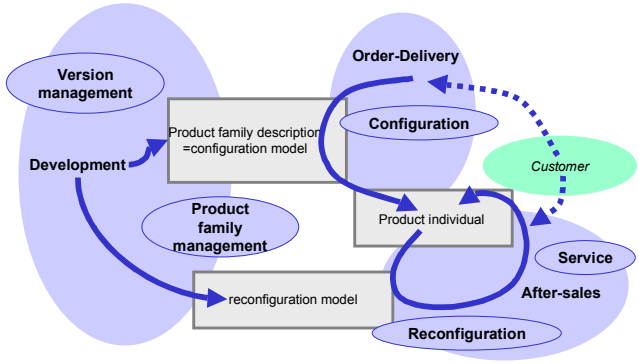


Figure 1. Basic product processes.

The generic software architectures, especially DSSAs, are close relatives to product configuration of traditional products. In both fields the term component means very much the same and also concepts 'port' and 'connector' are used, for example. In component based software, the components are very often seen as independent entities used in manner not thought at the time they were designed. In traditional products there are also such components, e.g., screws, bolts, capacitors and resistors. These, however, are typically not relevant to product configuration, in which components are larger chunks, also called modules, designed with the particular product family in mind. Of approaches to software variety, the closest to product configuration is modularization.

Furthermore, in DSSAs, the reference architecture with component library corresponds to configuration model and application configuration method to configuration algorithm of traditional products. Configuration models of traditional products are expressed by special languages designed for configuration purposes, which have the similar basis as ADLs of software architectures. Configuration languages, however, are typically applicable to all configurable products; they are not targeted to specific type of configurable products. It is not clear what are the right concepts for modeling software product families in general. Are components and first-class components the right concepts [18]; or should the configuration model capture the dynamic behavior of the system [19]; or should the configuration model be based on business processes [20]?

It would be in principle possible to model the physics behind the design of a traditional product. For example, one could include kinematics and fluid or energy flow equations and constraints in the

configuration model. This, however, is hardly ever done, as most companies do not have resources to build configuration models from physical principles Consequently, the configuration knowledge is typically at surface, design level. Thus, the fact that a configuration model describes only correct product individuals cannot be derived from the model—it is based on the designers' capabilities of designing functioning product families. The development of some software architecture styles in the form of general theory of software engineering resembles the approaches to develop a general theory of design.

5 PROPOSAL FOR A RESEARCH AGENDA

Our proposal is based on lesson learned in research with traditional products. The modeling and management of configuration knowledge of traditional products is difficult even with a static information. That is, for example, without analyzing whether a product individual fulfils some kinematic conditions. For configuring software, capturing the behavior of software has been proposed [19]. That is an important line of research, but unlike it, we begin here with a static situation. Regarding configuration of software product families, this means that we do not suggest starting from the general theory of software architectures. We thus intend to investigate a small part of the software architectures, which includes research on architecture description languages and domain specific software architectures in a context where large variety is central. Our work belongs to an emerging research area of software product lines in which the first international conferences are currently being organized (see http://www.sei.cmu.edu/plp/conf/SPLC.html). Our idea is to approach software configuration with the methods and tools developed for mechanical products. We aim to analyze how software products can be treated with them.

Our approach to managing software product families assumes

- a need for customer specific adaptation in a relatively routine configuration process. For example, because of restrictions in the size of the available memory.
- the software product to consist of components (or modules) that have clear interfaces
- existence of domain specific software architecture, or in other words, a configuration model, that describes the variants of the software product family
- a language for modeling the above mentioned components and the architecture, i.e., a configuration modeling language.

The management of a software product family would be done independently of the details of the process producing the software. This process may include selection of correct software modules, setting values for pre-processor directives, textual means (e.g., macros) for modifying the source code, selecting module versions from version management tools, creation of scripts for compiling and linking the executable, etc. The point is that the management is based on a configuration model at the architectural level. That model serves as a tool for the development and management of product family and for the actual configuration of software product individuals. The research tries to provide answers to, e.g., the following questions:

- How should the architectures and components of software product families and their evolution be modeled?
- What kind of intelligent support for re-using architectures and components and configuring software can be offered?
- How does the (dynamic) reconfiguration affect the situation? What does it enable?

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from Technology Development Centre of Finland (Tekes) and Helsinki Graduate School of Computer Science and Engineering (HeCSE).

REFERENCES

[1] K. Kumar and J. van Hillegersberg, 'Enterprise resource planning—experiences and evolution', *Communications of the ACM,* **43**, 22–26, (2000).

[2] S. Mittal and F. Frayman, 'Towards a generic model of configuration tasks', *in: Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI),* 1395–1401, 1989.

[3] T. Männistö, H. Peltonen, and R. Sulonen, 'View to product configuration knowledge modelling and evolution', *in: Configuration—papers from the 1996 AAAI Fall Symposium (AAAI technical report FS-96–03)*, B. Faltings and E.C. Freuder, eds. AAAI Press, 111–118, 1996.

[4] T. Darr, D. McGuinness, and M. Klein, Special Issue on Configuration Design. *AI EDAM* **12,** (1998).

[5] B. Faltings and E.C. Freuder, Special Issue on Configuration. *IEEE intelligent systems & their applications* **13,** (1998).

[6] H. Peltonen, T. Männistö, T. Soininen*, et al,* 'Concepts for modelling configurable products', *in: Proc. of the Product Data Technology Days,* Quality Marketing Services, 189–196, 1998.

[7] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a General Ontology of Configuration', *AI EDAM,* **12**, 357–372, (1998).

[8] A.W. Brown and K.C. Wallnau, 'The current state of CBSE', *IEEE Software,* **15**, 37–46, (1998).

[9] M. Shaw, R. DeLine, D.V. Klein*, et al,* 'Abstractions for software architecture and tools to support them', *IEEE Transactions on software engineering,* **21**, 314–335, (1995).

[10] D. Garlan and D.E. Perry, 'Introduction to the special issue on software architecture', *IEEE Transactions on software engineering,* **21**, 269–274, (1995).

[11] M. Moriconi, X. Qian, and R.A. Riemenschneider, 'Correct architecture refinement', *IEEE Transactions on software engineering,* **21**, 356–372, (1995).

[12] G.D. Abowd, R. Allen, and D. Garlan, 'Formalizing style to understand descriptions of software architecture', *ACM Transactions on software engineering and methodology,* **4**, 319–364, (1995).

[13] J.J.P. Tsai, A. Liu, E. Juan, and S. Avinash, 'Knowledge-based software architectures: acquisition, specification, and verification', *IEEE Transactions on knowledge and data engineering,* **11**, 187–201, (1999).

[14] P. Kogut and P. Clements, 'Features of architecture description languages', *in: Proceedings of Software Technology Conference,* 1995.

[15] B. Hayes-Roth, K. Pfelger, P. Lalanda, P. Morignot, and M. Blabanovic, 'A domain-specific software architecture for adaptive intelligent systems', *IEEE Transactions on software engineering,* **21**, 288–301, (1995).

[16] A. Karhinen, A. Ran, and T. Tallgren, 'Configuring design for reuse', *in: Proceedings of International Conference on Software Engineering, ICSE'97,* 701–710, 1997.

[17] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson, 'Applying software product-line architecture', *Computer,* **30**, 49–61, (1997).

[18] J. Bosch, 'Evolution and composition of resusable assets in product-line architectures: a case study', *in: Software architecture*, P. Donohoe, ed. Kluwer Academic Publishers, 321–339, 1999.

[19] C. Kühn, 'Requirements for configuring complex software-based systems', *in: Configuration—Papers from the 1999 AAAI workshop*, B. Faltings, E.C. Freuder, G.E. Friedrich and A. Felfernig, eds. AAAI Press, 11–16, 1999.

[20] A.-W. Scheer and F. Habermann, 'Making ERP a Success', *Communications of the ACM,* **43**, 57–61, (2000).

Yet Another Approach to CCSP for Configuration Problem

Mathieu Veron [1,2] and **Michel Aldanondo**[3]

Abstract. Industry interest concerning the configuration problem is getting bigger and bigger each year. Among the research axes raised by this activity, the search for a suitable formalism for configuration problem led to the definition of Conditional Constraint Satisfaction Problem. A CCSP could be briefly described as an auto-modifying CSP, variables and constraints are added or removed according to conditions on the current state of activity of variables. This leads to two separate types of constraints and to a two level algorithm.

We believe that the modeling of a configurable product must include and manage an activity state over each object used in the representation of such a product. So we are inclined to use the CCSP framework. But, there is a certain amount of drawbacks in using this approach. After a short summary of the main problems encountered, we will present another approach, able to represent the same class of problem but in a single CSP.

The choices made for this modeling will be justified and illustrated by an example. Furthermore, we will present an algorithm to solve this problem, this algorithm is able to take advantage of the nature of variable. To conclude, we will comment some preliminary results computed on real and randomly generated configuration problems.

1 Introduction

Nowadays, Constraint Satisfaction Problems framework (CSP) is used widely in industrial applications. Among the emerging ones, we would like to stress the growing importance, from both industrial and scientific points of view, of the configuration problem.

Configuration aims to provide a correct and complete definition of a product containing non predefined and/or optional elements. Computers, hand-made bookshelves, travels are examples of configurable products. The CSP framework can help us tackling this problem, first, thanks to its compact representation of the problem. Indeed, suppliers of configurable product generally have a wide offer and making an exhaustive catalogue of all available configurations is out of concern. Second, due to technical, commercial or even marketing decisions, some elements could not be combined together and should be identified at the designing stage, thus the framework used to represent a configurable product should be able to express constraints.

Earlier works [8] have shown that, if CSP seems to be a valuable candidate to manage the configuration problem, it could not been used ”as-is”.

[1] Access Commerce, rue Galilée,BP 555, 31467 Labège, France. email : mv@access-commerce.com
[2] Ecole Nationale d’Ingénieurs de Tarbes, 47 ave. Azereix 65000, France.
[3] Ecole des Mines d’Albi, Campus Jalard, Route de Teillet, 81013 ALBI CT Cedex 09, France. email : Michel.Aldanondo@enstimac.fr

1.1 Link between CSP and Configuration

Why CSP are able to represent a configuration problem. Configuration problems are often partitioned into two approaches : a technical one, where the configurable product is defined by the expression of all the parts that can compose the final product ; hence the configuration process could be seen as the reduction from this model to a valid bill-of-material (B.O.M.). Variables either represent the presence (or absence) of a component or its identification code. Constraints describe the allowed combinations of items. Moreover, one can express constraints over quantity of an item if there are variables dedicated to represent the amount of each selected item.

The second approach, said functional, starts with a more descriptive model of the product, more suited to the end-user lacking technical knowledge, it allows people to express their needs valuating characteristics without referring to parts. Variables can be used to represent characteristics, their domains and the possible values, whereas constraints can express the values which are compatible and those which are not. Moreover, one can express constraints over numerical values, for example dimension, which is very useful for tailored products. A mapping between the characteristics and the B.O.M. is then established.

Why they are not. First of all, there is no structure into a bare CSP whereas configurable product knowledge is quite always organized into a hierarchy. Hence, if variables are used to represent characteristics or the presence/absence of a component, they should be gathered into a meta-object, which could also be included in a super-element and so on.

Second, and this is one of the major differences between CSP and Configuration problem, all the variables do not take part in the solution. Indeed, the expert knowledge about configurable product is full of ”rules” such as : if a component of type A is selected then no component of type B should be available ; taking one of the component A implies to take one of the component C, etc. These rules can be conveniently coded through an activity state over variable, stating that only active variables belong to the search/solution space.

From these two statements, it becomes clear that not only variables activity should be controlled, but also that an activity state has to be managed at any level in the structure.

To overcome the above limitations, two main ways were explored : to manage on top of a CSP, a second problem able to compute the activity state of each variable (work published in [4]) ; or to extend the CSP framework in order to be able to express the structure of the problem, one of the most representative works was published in [5]. We will briefly discuss both approaches, and from the lessons we learnt, we will present another approach of the configuration problem modelization and the algorithm that comes along to solve it.

1.2 DCSP

Their works on configuration led Mittal and Falkenhainer [4] to present the Dynamic CSP formalism, also known as Conditional CSP (CondCSP) to distinguish from the Dynamic CSP presented in [2]. The idea behind CondCSP is to manage an activity state on each variable, and to allow the expression of rules to condition this activity. Only active variables take part in the solution. It could be formally defined as :

Definition. *A Conditional CSP is a triplet* $\{V, AC, CC\}$*.Where V is a set of variables, AC is a base of rules as "condition implies variable v is (or is not) active", condition is a logical expression involving either activity state or value of variables. CC is a set of constraints i.e. a subset of the Cartesian product of the domain of variables expressing the allowed valuations.*

The configuration problem is then defined as :

Definition. *The configuration problem is, given a CondCSP, to find a valid assignment such that all active variables satisfy all the constraints having all their variables active.*

Beginning with the CSP p_0 = $\{V_0, CC\}$, $V_0 \subset V$ is the subset of initially active variables (i.e. for which a rule like $true \rightarrow active : v$ exists)

1.3 Composite CSP

Composite CSP (CompCSP) were introduced by [5], as an answer to the lack of structure into the CondCSP formalism. The idea behind CompCSP relies on abstraction. At a meta-level, the problem is only composed of meta-variables, when they are selected, the whole problem is modified as the meta-variable is expanded into (and replaced by) a less abstract CompCSP.

Definition. *A composite CSP (CompCSP) is a triplet* $\{X, D, C\}$*, where X is a set of variables, D is the set of their respective domains and C is a set of constraints. Elements of any* $d_i \in D$ *is either a value or a CompCSP.*

Every time the domain of a variable v is restricted to a singleton representing a CompCSP p'={X', D', C'}, the current problem is modified : p_i becomes $p_{i+1} = \{X/\{v\} \cup X', D/\{d_v\} \cup D', C/\{C_v\} \cup C'\}$ where C_v is the subset of C containing constraints involving variable v.

We believe that this substitution mechanism is the strength and weakness of the CompCSP formalism. Indeed it allows to take care of the structure of configuration problems and to keep the current problem small. But, it introduces some drawbacks.

Due to the variable substitution, undoing a choice (i.e. backtracking) is more complicated than in classical CSPs. One needs to keep track of the meta-problem structure to undo choices and to remove variables and constraints. Moreover, in an interactive configuration framework, we need to keep all the choices made by the user "on screen". It is possible to implement this functionality in the CompCSP framework, but doing so, we loose the good small size property.

Regarding storing space and computing time issue now, some overcost due to "implied constraints" extrapolated from constraints given at the variable level might appear. Symmetrically, constraints expressed at the meta-level should be specialized at sub-level because, when the substitution occurs, all the constraints on this variable are removed from the new problem.

2 Modeling the Configuration Problem on a single CSP

In order to take the best of both worlds (expressiveness and structure), another definition of a configuration and configurable product will be given, then we will propose a way to encode it into a CSP.

2.1 Proposed model of the configuration problem

Let us start with the core definition, explanations will follow :

Definition. *A configurable product (CP for short) is a triplet* $\{L, V, C\}$ *where,*
L *is a set of CP, which can be partitioned in* $L = L_{cp} \cup L_{cp_e}$ *respectively containing non-elementary CPs (*$L \neq \emptyset$*) and elementary ones (*$L = \emptyset$*). Cardinality of* L *is* n*.*
V *is a set of variables, which can be partitioned in* $V = V_s \cup V_b$ *respectively containing state variables and base variables (i.e. the classical ones). Cardinality of* V_b *is* m*, whereas cardinality of* V_s *is* $n + m$*.*
C *is a set of constraints that can be partitioned in* $C = C_s \cup C_b \cup C_{sb}$*, respectively containing constraints involving only state variables, constraints involving only base variables and constraints involving both types of variables.*

The structure is given by the recursive inclusion of CPs within CPs. It is commonly admitted that keeping this structure tree shaped is safer. We encourage the use of a restriction rule within the definition not to allow circular inclusion of CPs. In the following, we will assume that all the manipulated CPs satisfy this condition.

State variables are Boolean variables, one for each CP and variable recursively included into the CP root. By definition, the value $\{0\}$ encodes the inactive state, while the value $\{1\}$ encode the active one.

Conditional rules over activity state are transposed into constraints involving state variables, for example : two elements A and B (either CP or variables) mutually excluding each other, or, if the use of A implies the use of B, see table 1 for examples.

Table 1. Condition Rules encoding

A excludes B

$State(A)$	$State(B)$
0	0
0	1
1	0

A implies B

$State(A)$	$State(B)$
0	0
0	1
1	1

both A and B or none

$State(A)$	$State(B)$
0	0
1	1

Whereas the CondCSP formalism used a two-level model and a two-level algorithm, we are now able to use CSPs classical algorithms such as propagation and resolution to solve the configuration problem. The advantages are numerous :

- It is often more compact than a rule based approach to encode the activity rules into constraints.

- Due to the attachment of constraints to CPs, the knowledge base itself is structured and CP re-utilization is easy.
- The good small-size property of CompCSP remains because only active variables participate in the solution and the resolution process.
- The transformation from CondCSP to this new modeling approach can be automated.
- It is possible to express the optional characteristic of a CP or variable (the one for which 0 and 1 are still valid into the domain of their state variable).

But more important is that, the same efficient algorithm is used for all kinds of variables, so we are able to :

- Deduce active and inactive CPs or variables thanks to constraint propagation.
- Explanation mechanisms [7] can be extended at no cost from value removal to state enforcement.
- The variable partition allows to decompose the constraint problem into three sub-problems (P_s, P_b, P_{sb}) on which we can enforce a different level of consistency.
- A resolution algorithm, while listing all the solutions of the configuration problem can exhibit the values that never appear into a solution. This can be done in a preprocessing step to obtain an equivalent and reduced problem.

2.2 Algorithm

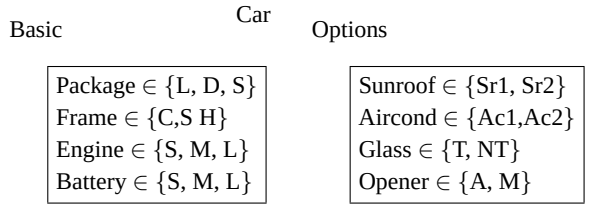
We will propose in this section, an algorithm to solve the configuration problem. It is based on Dynamic Backtracking [3] + local consistency, but contains some modifications. First the three sub-problems can be treated in a row, with different levels of consistency (available consistency are : one pass arc consistency (i.e. Forward Checking (FC)), Arc-consistency (AC) and Singleton Arc-Consistency (SAC)). Second, as constraints involving at least one non-active variable are trivially satisfied (in that case the constraint itself is said inactive), the propagation only occurs on active constraints. And last, a variable becoming active (respectively non-active) should be put in the propagation (resp. relaxation) pipe.

```
main program

i=0
definition of problem Pi
partition of Pi in Pis, Pisb and Pib
Filtrate each constraint of Pi
if inconsistent then exit on error
else
loop
    if in interactive mode
        wait for a user unary constraint
        (i.e. a choice of a value (val)
         over variable (var))
    else
        if in batch mode then
            select a variable (var)
            select a value (val)

    i++

    Filtrate(Pi, var, val)
        if Pi is not consistent then
            if in a batch mode then
                backjump, heuristically prefer
                 state variable
            else if in interactive mode then
                 explain inconsistency
                 and/or propose a restoration
end loop


function Filtrate(Pi, var, val)

push var into list L
while L is not empty
    v = pop(L)
    if v is in Pis  or Pisb
        filtrate domain of v (rather with AC or SAC)

        if the domain evolves then
            push the variables concerned into L
            if v is a state variable
             and
            if domain now equal to {1} then
                push the corresponding base
                variables (if any) into L
    if v is in Pib
        filtrate domain of v (rather FC or SAC)
        if the domain evolves then
            push the variables concerned into L
end while
```

2.3 Example

The example we have chosen here is the classical car configuration example ([4], [6]) enhanced with a small structural decomposition. the whole problem contains eight variables. Four of them compose the basic requirements, whereas the last four compose the options.

Figure 1. Example problem : car configuration



The problem is composed of eight base variables and three CPs. To encode the activity state, we need one state variable for each, so eleven state variables are required.

The table 2 shows how to encode the hierarchical composition of CPs within CPs and variables within elementary CPs. We can notice that two approaches are available : either one n-ary constraint like C1 or C2, or either many binary constraints like C3 to C5. the first approach is interesting when the constraint is very tight, the more extreme case is ”if the CP exist then all sub CPs exist” leading to two n-uplets all zero or all one. The second approach is more interesting when many sub-CPs are optional, the constraint is then very loose (one could have coded this by an n-ary constraint containing the forbidden valuation).

Table 2. Example of problem modelization

Constraints encoding the structure relationship.

C1 : Car is composed of Basic and Options		
state(Car)	state(Basic)	state(Options)
0	0	0
1	1	1

C2 : Basic is composed of Package, Frame, Engine and Battery				
state(Basic)	state(Package)	state(Frame)	state(Engine)	state(Battery)
0	0	0	0	0
1	1	1	1	0
1	1	1	1	1

C3 Options is composed of Sunroof	
state(Options)	state(Sunroof)
0	0
1	0
1	1

C4 Options is composed of Aircond	
state(Options)	state(Aircond)
0	0
1	0
1	1

C5 Options is composed of Glass	
state(Options)	state(Glass)
0	0
1	0
1	1

C6 Options is composed of Opener	
state(Options)	state(Opener)
0	0
1	0
1	1

Constraints encoding the former Activity Constraints, expressing the forbidden values (C7 to C15 from right to left and from up to down)

state(Sunroof)	state(Glass)
1	0
0	1

state(Engine)	state(Battery)
1	0

state(Opener)	state(Sunroof)
1	0

Package	state(Sunroof)
L	0
D	0

Package	state(Aircond)
L	0

Sunroof	state(Aircond)
Sr1	0

Frame	state(Sunroof)
C	1

Battery	Engine	state(Aircond)
S	S	1

sunroof	state(Opener)
Sr2	0
Sr1	1

Constraints encoding the former Compatibility Constraints, expressing the forbidden values (C16 to C19 from right to left and from up to down)

Opener	Aircond	Battery
A	Ac1	S
A	Ac1	L
A	Ac2	S
A	Ac2	M

Sunroof	Aircond	Glass
Sr1	Ac2	T

Package	Aircond
S	Ac2
L	Ac1

Package	Frame
S	C

Example of an interactive configuration.

1. At the beginning, and after a simple arc-consistency enforcing, Basic is active, and all the variables inside are active too. Battery is active due to the combination of C2 and C8
2. The user excludes L from the domain of $Package$ ($Package \neq L$), it remains two possible values for $Package$.
3. The user selects $Package \neq S$, by propagation, $sunroof$ and $Glass$ become active (by C10 and C7), so $Options$ becomes active (by C3 or C5) and $Frame \neq C$ by C13).
4. The user selects $state(Aircond) \neq 1$, this is a manual exclusion of a base variable from the search space.
5. The user selects $Frame \neq H$, $Engine \neq S$, $Glass \neq NT$, $Opener \neq A$
6. At this point we let a tree search algorithm complete the configuration for us, leading to $Engine \neq L$, $Battery = S$.

Discussion Looking at step 1, we can deduce that the model could have been simplified suppressing C8 and the second tuple of C2.

Step 3. $Frame\{C\}$ has been removed by propagation, but a complete pre-processing of the model (for example, using [1] we are able to compute the whole set of solutions) would have detected that $Frame\{C\}$ should be removed from the model. Thus it is possible to reduce the problem to an equivalent one, suppressing this value from the domain and the tuples using it.

Step 4. Enforcing a state value is an elegant way to solve the problem of user choice over an optional variable, at this stage, $Aircond$ could still become active or not, the user has decided to have no air conditioner. Thanks to the representation of state through a variable this choice is propagated the same way as classical valuations.

Step 6. The user think he has expressed his needs, and do not care of the exact values that are going to be chosen for the remaining variables. A tree search algorithm is then run on the problem consisting of active not-instantiated variables (domain not restricted to a singleton), trying to extend the current valuation to a valid solution (i.e. a configuration) without backtracking on user choices. This feature is very useful within configurators.

3 Conclusions

The main contribution of this paper to the configuration domain is to propose a model able to express a configurable product in a hierarchical way which is compatible with most of the CSP algorithms. As far as we know, only the work reported in [5] has dealt with this hierarchical approach problem which is a fundamental aspect for configuration deployment in industry.

The proposed elements have been set in a software package and tests on various problems (including randomly generated ones) have been conducted. Time measurements on interactive configuration with rather small models (fifteen state variables, fourteen base variables) is under the tenth of second, deserving the title of interactive. Achieving various levels of consistency over the sub problems does not show, for now, significant results. In our opinion, more precise parameters of a configuration problem (and of randomly generated problems) should be studied, in order to show their influence over various search algorithms. This is an ongoing work we will focus on, as well as adapting optimization techniques to our model.

ACKNOWLEDGEMENTS

We would like to thank Access Commerce, its management and R&D teams for their support all along this research work.

REFERENCES

[1] Amilhastre J. : Représentation de l'Ensemble de Solutions par Automate d'Etats. Thèse de l'Université de Montpellier, (1999).
[2] Dechter R., Dechter A.: Belief Maintenance in dynamic constraint networks. In proceedings of AAAI'88, (1988), pp. 37-42.
[3] Ginsberg M. : Dynamic Backtracking, in Journal of Artificial Intelligence Research. AI Access Foundation and Morgan Kaufmann Publishers, (1993), pp. 25–46.
[4] Mittal S.,Falkenhainer B.: Dynamic Constraint Satisfaction Problem. In proceedings of the 8th AAAI conference, (1990), pp. 25-32.
[5] Sabin D., Freuder E. : Configuration as Composite Constraint Satisfaction. Technical Report FS-96-03, AAAI Fall Symposium on configuration, Boston. AAAI Press (1996), pp. 28–36.
[6] Sabin M., Freuder E. : Detecting and Resolving Inconsistency and Redundancy in conditional Constraint Satisfaction Problems. Workshop AAAI'99 on configuration, Orlando. AAAI Press (1999).
[7] Verfaillie G., Lobjois L. : Problèmes de satisfactions de contraintes. Revue d'Intelligence Artificielle, volume 9, number 3 (1995), pp. 339–373.
[8] Veron M., Fargier H., Aldanondo M. : From CSP to Configuration. Workshop AAAI'99 on configuration, Orlando. AAAI Press (1999).

Product Data Management (PDM) System Support for the Engineering Configuration Process

- A Position Paper -

Samir Mesihovic[1] and Johan Malmqvist[2]

Abstract. This position paper treats the use of a PDM system integrated product configurator to support development and configuration of highly customized product variants. The problem is investigated as a part of a Ph.D. project performed at Chalmers University of Technology, Sweden.

1 INTRODUCTION

During recent years, competition between engineering companies has become much harder. The companies that win the competition are those who can first deliver highly customized products that meet customer requirements and company constraints.

PDM (Product Data Management) systems and product configurators are computer tools that make it possible for companies to shorten product development time and sales-delivery process. The term sales-delivery process includes all the phases required to sell, design, order, manufacture and finally deliver an individual product to a customer [4].

PDM systems keep track of the masses of data and information required to design, manufacture and then support and maintain products during the entire product life cycle. PDM systems also provide support for modelling of processes that can be executed on the data in the system [2]. *Product configurators* are systems that use product definition information in the sales-delivery process for fast and correct configuration of working product variants that fulfil customers' requirements and company constraints such as production and delivery capabilities.

The problem is that the product configurators as used in many companies these days, do not fully support the engineer-to-order process that sometimes needs to be done in order to further customize a preliminary configured product (Chapter 2.2). This task demands an integration between product configurator and engineering applications such as CAD/CAM/CAE.

1 Department of Machine and Vehicle Design, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, e-mail: samir@mvd.chalmers.se

2 Department of Machine and Vehicle Design, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, e-mail: joma@mvd.chalmers.se

Another problem is to update product configurators with a new release of product configuration data. Invalid information in the product configurator systems leads to incomplete and invalid product specifications. For this reason correct information in the product configurators is vital for quality and assurance. Moreover, the engineering staff that has most knowledge about the company's products can often not transfer that knowledge to the company's configurator because there is a lack of user-friendly methods and computer tools for such task..

A more effective product configuration knowledge transfer from the product development process to the sales-delivery process and product configurator is needed.

Different types of the product configuration concept are presented in chapter 2. Chapter 3 gives an introduction to the PDM system integrated product configuration concept and an example is presented. Finally, conclusions and recommendations for future work are given in chapter 4.

2 PRODUCT CONFIGURATION

Mittal and Frayman define product configuration as a special type of design activity, with the key feature that the artefact being designed is assembled from a set of pre-designed components that can only be connected together in certain ways [6].

Pre-designed components are re-usable, completely designed in detail and can be manufactured without any additional information [3, 4].

Up to now, product configurator systems have been mostly implemented as commercial sales configurators, as parts of ERP systems or as company specific developed solutions.

These configurators work well for products that exclusively consist of pre-designed components. However, in some cases, products consist of a mix of pre-designed, parametric and modifiable components, e.g., speciality valves used in the process industry.

Parametric components are pre-defined to some extent, but the key parameter values must be determined before they can be manufactured and assembled [3].

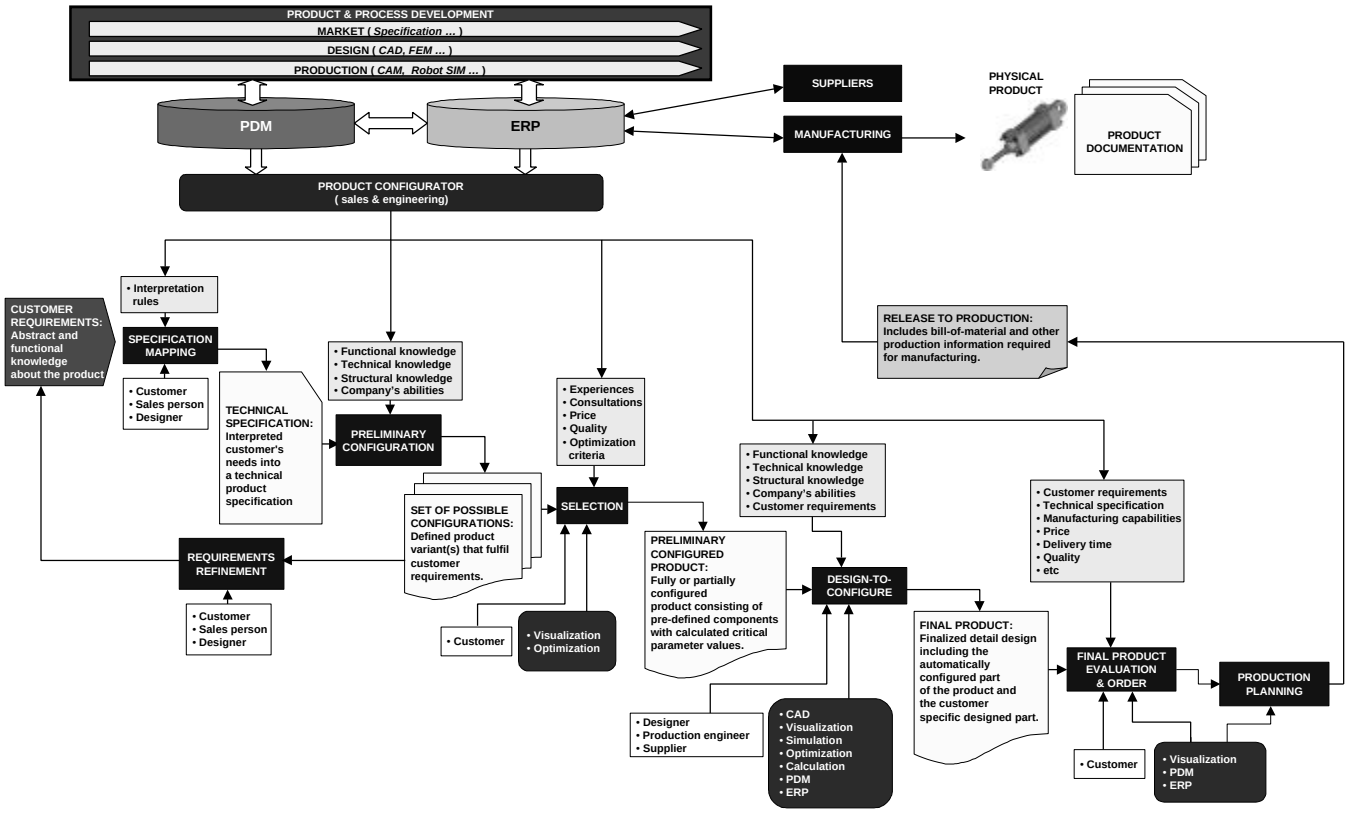


Figure 1 Sales-delivery process containing an automated product configuration part where the pre-designed components are identified and assembled, and an engineering part where the initially configured design is finalized. The general model of integrated product and process development is re-drawn from Andreassen and Hein [1]. The procedure for product configuration is an extension of Schwarze's [7] model by adding the design-to-configure step.

A modifiable component has pre-defined functionality, but the design is not solved in detail because of the large variations in customer demands when it comes to that part of product's functionality, e.g. inlet and outlet connector design for high pressure reducing valves used in the process industry. In order to define a modifiable component, an engineering task must be done during the sales-delivery process.

The existing configurators provide limited support for such products. For this reason, the definition of configuration, given above, should be extended in order to support configuration of products containing pre-designed, parametric and modifiable components. It follows that two product configurator types / product configuration strategies can be identified:

- Assemble-to-order configurators supporting traditional assemble-to-order process.
- Engineer-to-order configurators supporting engineer-to-order process.

2.1 Assemble-to-order

Assemble-to-order configurators have standard configuration abilities presented by Mittal and Frayman. They give strong support for the assemble-to-order process but have limited capabilities for supporting the engineer-to-order process. Product configuration models [4] used in the assemble-to-order configurators are typically built after the product development process and are based on a fixed number of components and rules for their assembly. That results in a fixed number of product variants that can be offered to the customers (computers, most automobiles). The product structure generated during the product configuration process often can not be modified afterwards. Furthermore, assemble-to-order configurators are often integrated with order and manufacturing systems so that once they have been configured, product variants can automatically be manufactured and delivered. Assemble-to-order concept presented above includes typically the specification mapping step, the preliminary configuration step and the selection step of the sales-delivery

process (Figure 1). After the selection step, release to production, manufacturing and delivery can be automatically proceeded.

2.2 Engineer-to-order

In the engineer-to-order configuration concept, some parts of the product are pre-designed components and can be automatically configured while other components must be specially designed according to customer requirements (transformers, speciality valves, elevators). The automated part of the engineer-to-order process can use standard assemble-to-order configuration approach. The engineering part of the process has to be supported by a configurator but also by other applications, e.g CAD/CAM/CAE tools in order to assure a fully working end-product (see Figure 1).

During the automated configuration part of the engineer-to-order process an "open" product structure is generated. In the open product structure some objects are fully identified by using pre-designed components, including article numbers and assembled in certain way, while other objects are left unidentified and an engineering effort needs to be done in order to define this part of the product structure [5].

The unidentified objects of the product structure are typically parametric and modifiable components. Parametric components may be treated as pre-designed components if the parameter values are determined during the automated part of configuration process by using calculation applications directly integrated with the assemble-to-order configurator. However an engineering process must be done in order to design the modifiable components. In this case, the configurator can deliver information concerning modifiable components, e.g. customer requirements, component function, component type, parameter data, and component template, that helps engineers to effectively design the solution. This requires support from CAD/CAM/CAE tools. The engineering part of the engineer-to-order process is here called design-to-configure (Figure 1). An important constraint for the design-to-configure process is that pre-designed, parametric and modifiable components assembled in the end-product have to work together properly to achieve functionality according to the customer requirements. Furthermore, the workflow and the product configuration data has to be managed in order to ensure that important engineering phases are followed and that the data needed is available in every step. This can be supported by using a PDM system. It is also important to have information about production, delivery and supplier abilities during the design-to-configure process. This demands an integration to the ERP system.

The engineer-to-order configuration concept enables companies to dramatically increase the number of available product variants on the market and is close to the custom engineered product concept (prototypes), while using mass-production and assemble-to-order possibilities for the pre-designed components. The difficulty is then to do this in an efficiency equal to or higher than that of the assemble-to-order concept. There is therefore a need to manage the product configuration data in a more efficient way during the product development and the sales-delivery process. This issue is discussed in chapter 4.

3 PDM INTEGRATED PRODUCT CONFIGURATION CONCEPT

From the beginning, PDM systems were aimed at supporting all design engineering work, but have had weak support for product configuration. Improved capabilities for product configuration support have been introduced recently in major PDM systems such as Windchill, eMatrix, Metaphase, IMAN [8, 9, 10, 11]. These solutions give the user better facilities for developing product variants, product configuration models and product configurator portals in an integrated environment.

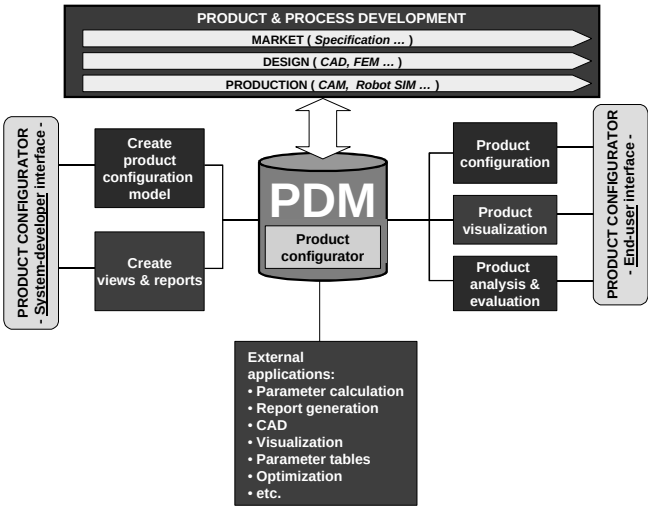


Figure 2 PDM integrated product configurator where the PDM is used for management of product configuration data during the product development process and their communication to the product configurator. Furthermore it provides integration with external CAD/CAM/CAE applications.

During the product development process, the PDM system is used to manage product data in different phases. At the same time, PDM enables concurrent development of products and product configuration models (Figure 2). A PDM integrated product configurator further provides support for version management of product configuration models. It enables the user to re-use old

product configuration models and retrieve old product variants. This is important for maintenance and delivery of spare parts. The configuration logic stored in a PDM integrated product configurator is typically parameters, options and constraints (explicit and case tables). For simple product configuration models, there is no need of specialist programming skills. A graphic representation makes the configuration modeling task easy. For more complex products a programming effort still needs to be done. That enables the system administrator to create more advanced product configuration models. It also supports integration with external applications such as parameter calculation programs, publishing programs, constant tables sheets, visualization tools etc. This can be important for companies which already have assemble-to-order configurators but need further product customization during the sales-to-delivery process. In this case, a PDM based product configurator can first execute an external assemble-to-order configurator and then proceed to a customization process for the automatically generated solution.

Many of the new product variants on the market are derived from existing product variants, where a part of the product is new designed and the rest is based on the old concept. By using a PDM integrated product configurator, engineers can test how the new concepts work together with old ones at an early stage. The engineer can also generate the old product variants and study them in order to approve the new ones. Product configurators contain all product information needed to effectively generate product variants that fulfil customer requirements. That information can be submitted to the engineer via the PDM, so that he can identify and report the product configuration data needed for configuration of the new product variants.

Our current work is concentrated on investigating the pros and cons of PDM integrated product configurator solutions. The research method applied is based on a combination of case studies at three Swedish engineering companies and the development of demonstrators in commercial PDM tool with integrated product configurator.

As a first result, a test system for configuration of hydraulic cylinder variants has been developed. The system is based on a PDM system integrated product configurator (Windchill Product Configurator 4.0), a CAD system (Pro/ENGINEER) and a mathematical software (MATLAB 5.3).

4 CONCLUSIONS AND FUTURE WORK

There is justification for a renewed investigation of the possibilities of implementing support for development of the configurable products in PDM systems. The idea is to manage the product configuration data in the PDM system during the product development process in such way that it is later easy to use in the sales-delivery process for updating the company's product configurator. Furthermore, it is important to support the design-to-configure process where the product configuration data is needed to effectively finalize once preliminary configured product according to the customer requirements.

Integration between PDM and ERP systems is still a bottleneck in companies, and it can lead to problems in updating PDM based configurators in relation to production and supplier capability data.

The expected result is planned to be a method for use of PDM tools to support management of product configuration data in the product development and the sales-delivery process for the mechanical products containing pre-designed, parametric and modifiable components.

5 ACKNOWLEDGEMNTS

This work was financially supported by NUTEK, the Swedish National Board for Technical Development. Special thanks also to a Swedish engineering company, BTG Källe Inventing, that develops and manufactures specialty valves for process industry. We would also like to thank people at PTC Nordic's office in Sweden for the support concerning Windchill and Pro/Engineer.

6 REFERENCES

[1] M.M. Andreassen and L. Hein, *Integrated Product Development*, Springer-Verlag, New York, 1987.

[2] CIMdata: *Product Data Management: The Definition, An Introduction to Concepts Benefits, and Terminology*, CIMdata Inc., Fourth Edition, 1997.

[3] J. Tiihonen, T. Soininen. T. Männistö, R. Sulonen, *State-of-the-practice in product configuration - a survey of 10 cases in the Finnish industry*, IIA Research Centre, Helsinki University of Technology, Finland, 1996.

[4] J. Tiihonen, T. Soininen, *Product Configurators - Information System Support for Configurable Products*, TAI Research Centre and Laboratory of information Processing Science, Product Data Management Group, Helsinki University of Technology, Finland, 1997.

[5] L. Jansson, *Business Oriented Product Structures*, Thesis for the Degree of Doctor of Philosophy, Department of Production Engineering, chalmers University of Technology, Sweden, 1992.

[6] S. Mittal, F. Frayman, *Towards a generic model of configuration tasks*, Proceedings 11th International joint conference on artificial intelligence, 1395-1402, USA, 1989.

[7] S. Schwarze, *Specification Mapping - the integration of customer requirements within a product configuration*, Institute of Industrial Engineering and Management (BWI), Switzerland, Swiss Federal Institute of Technology (ETHZ), 1993.
[8] PTC, *Windchill*, http://www.ptc.com/products/windchill/prodplanning/ds_factor.htm, 2000.
[9] MatrixOne, *eMatrix*, http://www.matrixone.com/products/applications.html, 2000.
[10] SDRC, *Metaphase*, http://www.sdrc.com/nav/software-services/metaphase/configure.html, 2000.
[11] Unigraphics Solutions, *iMAN*, http://www.ugsolutions.com/products/iman/products/, 2000.

Conceptual Modeling of Product Families in Configuration Projects

Niels Henrik Mortensen[1], **Bei Yu**[2], **Hans Jørgen Skovgaard**[2] and **Ulf Harlou**[1]

Abstract. During the early phases of configuration projects very important decisions are made which will heavily influence the performance of the company, benefits in different functional areas (production, sales, purchase, product development, service etc), maintenance of the configuration system and quality of the dialogue between the configuration system and the users. Today there exists very sparse tools and procedures which can assist the early phases, i.e. conceptual modeling of the products and product assortment. This paper presents a five-phase procedure for conceptual modeling in configuration projects. Each of the five phases is supported by a set of tools. The main idea of the procedure is utilization of a so-called Product Family Master Plan, which is a formal description of the product assortment and its variation. The procedure has been tested at one of Baan's customers with very convincing results.

1 GROWING PRODUCT ASSORTMENT

In many companies the product assortment is growing quickly due to an increased number of customer specific product variants. There are many "good" reasons for that, e.g.:

- When a customer is on the phone and wants a new feature included, it is difficult to say no.
- No one in a company dare to cut away variants.
- No one dare to say no to a great new product invention.
- Resources utilized for maintaining the product assortment is not measured and visible.

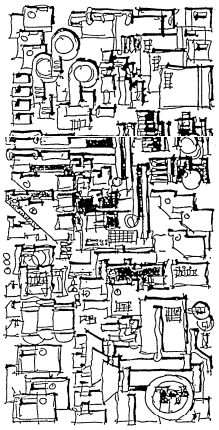


Figure 1. Complex product assortment

The growing product assortment normally leads to an increased turnover but not necessarily to increased profit. The reason is that all new parts or products become tasks in all functional areas in the company. Some one has to purchase, storage, produce, market, sell, ship etc. the new variant. The company might be trapped, thus the majority of resources are spent on maintaining the big complex product assortment and the ability to innovate and develop new product are dramatically reduced.

Now one could argue that it is not difficult to see the problems, but what about the solutions? A main challenge is to master the right balance between variance and commonality. The product assortment shall show variance from a market point of view, and show commonality from a company point of view. Variance of the product assortment is a relational property between the product assortment, competitor products and customers. Commonality is a relational property that at least two products have in relation to a functional area. If e.g. two machine parts can be manufactured by the same lathe then they posses commonality from production point of view. Commonality ensures reduction of complexity in all activities carried out in a company.

How shall a company handle the right balance between variance and commonality? There exist two very powerful alternatives, namely application of modularization and configuration. Modularization is a way of structuring the product assortment and configuration is way of handling knowledge about the product assortment. Configuration and modularization can be applied individually or in combination. Sometimes products are not configurable and therefor modularization is a prerequisite for obtaining a product assortment, which is configurable.

This paper is treating configuration – more specifically the conceptual phases in a configuration project. The next section will briefly explain the reasons for doing conceptual modeling in a configuration project, and then a procedure and toolbox for conceptual modeling will be presented. This procedure and toolbox has been tested at one of Baan's customers and experience from application will be explained.

The work is done within Baan Development, which is currently developing a new powerful language for configuration. This procedure for conceptual modeling is targeted at the new configuration language, [1], [6]. This modeling language from Baan will be so flexible that an approach taking the need of the company as starting point is possible rather than taking the staring point in the modeling language and tools.

[1] Department of Control and Engineering Design,
Technical University of Denmark, Building 358, DK-2800 Lyngby,
email: nhm@iks.dtu.dk

[2] Baan Company, Baan Nordic, Hørkær 12A, DK-2730 Herlev,
email: byu@baan.com, hjskovgaard@baan.com

2 REASONS FOR CONCEPTUAL MODELING

When a company has finished modeling of the product assortment very important decisions have been made. It is decided which product variants that shall be offered to the market and thereby which products and subsystems that shall be manufactured (purchase, logistics, transport, service etc). This means that both turnover and costs are influenced directly by the contents of the configuration system. It can therefore be argued that deciding on the structure of the configuration model is a *business decision* rather than a configuration technology decision. Making a constraints may seem innocent but may be a very crucial decision.

Like in any other projects important decisions are made in the early phases of a configuration project, i.e. the phases where the model structure in the configuration system is decided. Experience shows that when 10-15% of the resources are consumed 80% of costs, maintainability, efficiency and effectiveness etc are disposed.

Today there only exist sparse results from the early phases of a configuration project. Often there is a jump from specification to typing in the attributes, constraints, resources etc in the configuration system, see figure 2. This is an unfortunate situation because the domain experts, e.g. sales, production, purchase, product development have difficulties on seeing what decisions are made and they can not react.

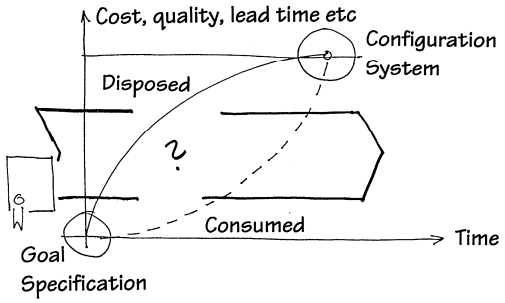


Figure 2: Sparse results from the conceptual phases

The main reason for developing a procedure handling conceptual modeling is to make the early phases explicit and visible. This is expected to improve conditions for bringing in relevant stakeholders early in the projects and improve conditions for ensuring that there exists a proper leitmotif from overall business goals to contents of the configuration model.

Another reason for developing the procedure is that it is sometimes difficult to maintain and update a configuration model when new products are introduced in the product assortment. It can be difficult to remember the meaning of attributes, constraints, resources and mode of action for the configuration system.

Last but not least it is the intention that the conceptual modeling procedure shall make it possible for a company to divide the modeling of production families between domain experts (purchase, production, quotation, sales, product development) and IT specialists.

To sum up there are four main reasons for developing a procedure for conceptual modeling of product families in the Baan configuration system:

- Support business decisions in the early phases of a configuration project.
- Improve documentation thus conditions for updating and maintenance of the configuration system is improved.
- Allow a possible labor division of modeling product families between domain experts (e.g. sales, purchase, production. design, product development) and IT specialist.
- Allow the starting point for modeling to be the need of the company rather than modeling language and tools.

3 HOW TO MODEL A PRODUCT ASSORTMENT?

According to systems modeling [7], [8] there are two types of attributes, which are relevant when a product or product assortment is being modeled. These attributes are named structural and behavioral attributes. Structure is answer to the question, what is it? and behavior is answer to the question, what is it able to do? Examples on structure attributes for a car are car type, size of the engine, number of doors and color. Examples of behavioral attributes for a car are speed, acceleration, noise-level and fuel consumption. The distinction between structural attributes and behavioral attributes is relevant because only the structural attributes can be determined directly during configuration. Formally speaking the behavioral attributes and the structural attributes are related to each other in a causal way. The identification of structure and behavioral attributes is fundamental when modeling products.

Another aspect that is relevant to modeling is deciding upon the overall structure of the configuration model. A product has several structures, see figure 3. A product has e.g. supply, purchase, production, assembly, shipping, transport, sales, maintenance and recycling structures.

Now one could ask which of the above structures should the configuration model be based on? Some of the factors that will influence the structure are application area of the configuration system, frequency of application, stability of the product assortment, domain complexity. Probably there does not exist a generic answer to determining the structure but often a functional structure is feasible. Seen from a front office point of view, customers are generally asking for solutions, which solve functions. Seen from a back office point of view production shall deliver parts which will be come a product with certain functionality. A functional structure means that each element in the model solves functions. Identification of the functional structure is supported by the functions-means tree law. This tool is described further in section 4.2.

It is relevant to work with more alternative product structures because one can not look at a product structure and ask is this good or bad. Only by comparing alternatives it is possible to evaluate and find the most suitable one.

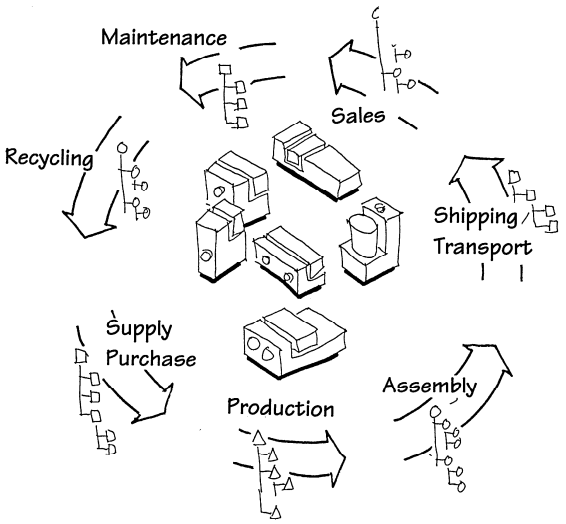


Figure 3 A product assortment has several structures, [5]

A formal way of describing the product assortment is a so-called Product Family Master Plan (PFMP). A PFMA consist of two main elements: a generic part-of structure and a generic kind-of structure, cf. figure 4.

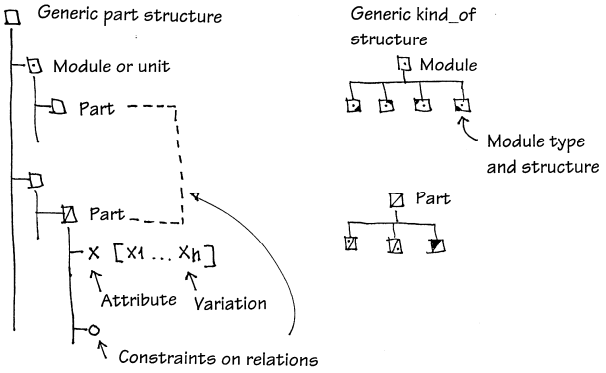


Figure 4: Contents of Product Family Master Plan

The generic part-of structure describes the modules, assemblies and parts that exist in all the products within the assortment or the product family. Each element is described by attributes that are determined during configuration. The Generic kind-of structure describes the modules, assemblies and parts which are changeable in the product assortment. The product family master plan is thereby a complete description of the product assortment and the way variants are created.

The PFMP is described on a big piece of paper typically 2 meters times 3 meters. Very often companies have never been able to get an overview of the whole product assortment, see figure 5.

Experience has shown that the product family master plan is very good tool for discussing the product assortment, i.e. where to start modeling in the configuration system, which variants shall be included, which technologies are stable and which technologies will change? Where does the company make money on the variants and where does it looses money etc. Having a complete and visual descriptions of the product assortment improves the possibilities for involving domain experts (sales, production, design) and company management in the crucial decisions during modeling in a configuration project.

Figure 5: Discussing the product assortment in front of the product family master plan

The product family master plan constitutes the core of a modeling procedure that is the topic of the next section.

4 MODELING PROCEDURE

The modeling procedure consists of five phases supported by a number of tools. The content of each phase is further explained in the next sub sections.

- Identification of configuration task.
- Identification of product family master plan.
- Conceptual modeling of product family master plan.
- Detailed modeling of product family master plan.
- Modeling of product family in the configuration system.

The amount space in this paper does not allow all tools to be described in details and therefore focus is put on the tools that has been developed within Baan and Technical University of Denmark.

4.1 Identification of configuration task

The purpose of this phase is to define what the configuration system must accomplish in accordance with stakeholders demands and overall business objectives. Important questions to be answered in this phase are: Who should benefit from application of the configuration system? Where to harvest the benefits?, Which activities shall be supported by the configuration system?, What are the leitmotif from creation of business to contents of the configuration system?

The tools supporting this phase are:

- Stakeholder identification
- IDEF 0 – activity analysis
- Scenario techniques

- Use-cases
- Life cycle analysis

4.2 Identification of product family master plan

The objective of this phase is to get an overview and describe the product assortment. The means for describing the product assortment is the Product Family Master Plan. Important questions that this phase should contribute to answering: What are the variants that the company wants to offer to the market what are the variants that the company does not want to offer? What variants can be expected in the future?

Figure 6 and 7 shows examples on working with a Product Family Master Plan.

Figure 6 Working on different alternative Product Family master plans

Figure 7 Final Product Family Master Plan

The identification of the function product structure is supported by the so-called function means tree, see figure 8. The starting point is identification of the overall function of the product. The overall function of an overhead projector (a means) is to enlarge and project an image. This function can be carried out by different means, e.g. slide projector principle, OHP principle and Episcope principle. Each of these means requires the existence of lower level functions, e.g. carry image, provide light, diffuse light, focus light, change direction, see figure 8. By this principle the function structure for a product assortment can be describe hierarchically.

The function means tree diagram provide an overview of alternative ways a function can be realized and is a good foundation for discussing the existing product assortment variants and identification of which part of the function structure that will be invariant and which parts is likely going to change.

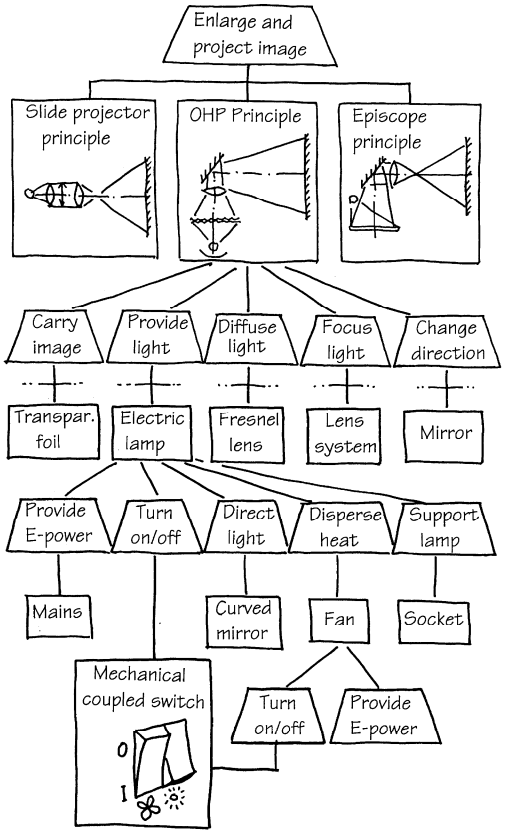


Figure 8 Function means tree, based on [4]

Supporting tools in this phase are:

- Terminology charts
- Function means tree
- Product family master plan chart
- Variance card
- Commonality card

4.3 Conceptual modeling

In this phase each of the elements of the product family master plan is described. For this phase a modeling tool has been developed [9], which is inspired by the object oriented modeling paradigm, [2] [3], see figure 9. In this modeling tool each class in the product family master plan is described a Class Description (CD) card. Below the contents of a CD card is further explained.

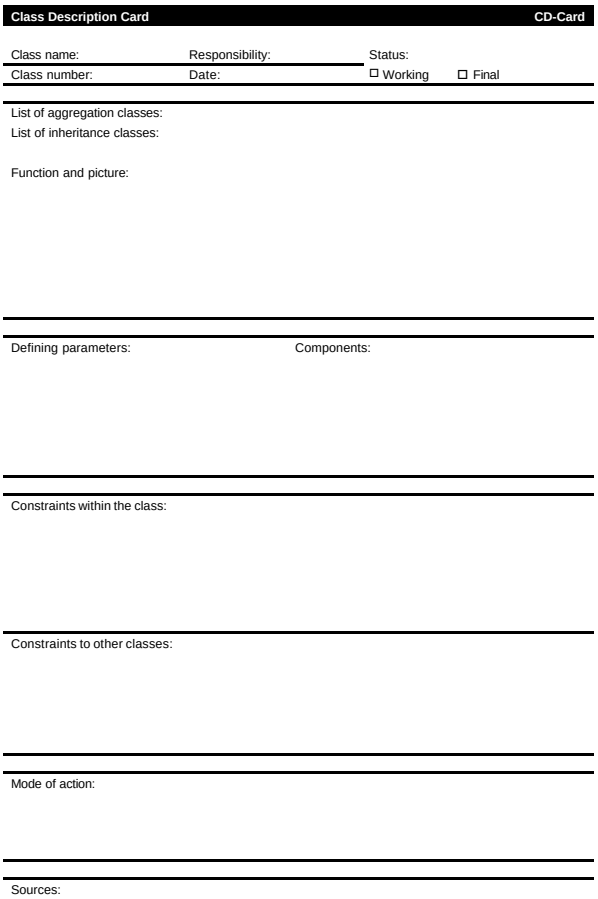


Figure 9: Class description card

List of aggregation classes describes part-of relations to other classes. Aggregation classes for a car could be engine, transmission system and chassis.

List of inheritance classes describes the kind-of relations to other classes. Inheritance classes of transportation equipment can be e.g. busses, planes and trains.

Function and picture contains a short description of the functionality of the class within the product. As en example the main functions of a window is allow light to enter, allow venting and allow people to escape in case of fire etc. Picture contains a sketch/picture/diagram of the class. This is relevant for deciding on a consistent product terminology. When e.g. geometry is involved this can be utilized for determining coordinate system conventions, governing parameters etc.

Defining parameters are the attributes that can be determined directly during configuration.

Components are elements, which the class consists of. For a car components might be sunroof, stereo and air-condition which are typically boolean and not further defined.

Constraints within the class are restrictions on how the components and defining parameters may be related. An example of a constraint within a car class could be that cars sold in South America should have an air conditioning system.

Constraints to other classes are restrictions on how the classes can be related to other classes in the configuration system.

Mode of action is description of the input and output parameters to the class. Input and output can be related to the user of the configuration system and other IT systems

Sources describe the reason for the contents of the CD Card. The idea is that all defining parameters and constraints shall have a reason. The reason can e.g. be a stakeholder, documents within the company.

4.4 Detailed modeling

In this phase modeling is carried out in the same way as conceptual modeling but the configuration language used in the configuration system is utilized. The previous phases are independent from choice of configuration system. Main supporting tool in this phase is CR (Class Responsibility) cards, which has the same structure as class description cards.

4.5 Modeling by means of configuration language

In this phase the model is build up and tested within the configuration system. It is possible to divide the modeling between domain experts and IT experts. It means that identification of configuration task, identification of product family master plan, conceptual modeling are carried out by domain experts and detailed modeling and modeling in the configuration system could be carried out by IT specialists if desired based on the Class Description cards and Class Responsibility Cards.

5 EXPERIENCE FROM APPLICATION

The modeling procedure has been tested in one company. The case company is developing, manufacturing and selling building equipment. Approximately 5.000 people are employed in the company, which produces 4.500.000 products pr. year. In order to stay competitive the company has to be able to deliver customized products to the end users. The intention is to be able to deliver more than 300.000 variants seen from a customer point of view. Currently the products are not configurable, due to complex plastic components that are not changeable. Therefor a so-called fulfillment project has been initiated in which 150 people are working. Approximately 25 people are working on the configuration part of the project.

The family master plan in this case was drawn on a big piece of paper approximately 4 meters times 2 meters. It was placed in the center of the project room where the 150 people were working, thus the majority of people working on the project had to pass the product family master plan on the way to their desks, to meetings etc.

During the project, this family master plan was important for creating dialogue with many kinds of stakeholders. Three alternative family master plans were developed during the projects. It

seems important to carry out alternatives because there does not exist one correct product family master plan. The different alternatives will lead to configuration systems that possess different performance, maintainability and flexibility to bring in new products in the assortment. The most important consequence of utilization of the product family master plan is that it is bringing a concept phase into a configuration project, thus the family master plan is visible and it is possible for stakeholders to react on the contents before modeling in the configuration system starts.

Reactions from this company are:

- "I like the naming convention that you have used throughout your documentation"
- "It is crucial to have an unequivocal naming convention and way of describing things"
- "The way of visualizing the product family master plan was beneficial and provided an overview of the product assortment that we did not have before"
- "Your sketches in the Class Description cards together with the product family master plan are GOOD"
- "Your documentation does not tell how the configuration model is maintained, but it is the foundation for being able to maintain the model. If these documents does not exist we can not maintain the configuration model"
- "If you had not introduced the Product Family Master Plan I doubt that we would have realized the commonalties between the product families"
- "It is difficult for us to forget the Bill Of Material. It is nice to see that the models are built up from function, functional units, assemblies and parts because we make decisions about these"
- "It sound like the Product family master plan and the Class description cards is great for getting our thoughts down"
- "We consider using this procedure for the Back Office Configuration too"
- "We need to train people in the very important thinking pattern that is behind this procedure"
- "The people who fill out the CR cards must have detailed knowledge about the product, manufacturing, market and some basic understanding of object oriented modeling"
- "This is the only procedure that I have seen for documenting the configuration model"
- "I like the product family master plan, because we can bring in relevant stakeholder earlier on"

6 CONCLUSIONS

It is very clear that handling the conceptual phase of a configuration project is crucial for the success. It is often difficult to handle the conceptual phases because the knowledge by definition is vague. Application of the proposed procedure based on the Product Family Master Plan is formalizing the modeling activities in the early phases and proposes a visual way of working. The implication of this is that it is possible to involve stakeholders earlier on and have “facts” based dialogue concerning variance and commonality of the product assortment.

The Class Description Cards ensure documentation and traceability of a configuration model. Preliminary experiences shows that is it possible to divide the modeling task between domain experts and IT specialists, thus that modeling in the configuration system can be carried out based on the contents of the Class Description Cards.

Further work on the conceptual modeling procedure involves application in two Baan configuration projects. The next step for the modeling procedure is development of a Baan course on conceptual modeling that will be launched together with the new Baan configuration language and tools. The goal of this course will be to train people in companies, thus they are able to model the product assortment by means of the modeling tools.

REFERENCES

[1] Baan, Baan CAVA Concept guide [Confidential], Baan Company, Herlev 2000

[2] Coad, P & Yourdon, E: Object oriented analysis, second edition: Prentice Hall, New Jersey 1991.

[3] Coad, P & Yourdon, E: Object oriented design, second edition: Prentice Hall, New Jersey 1991

[4] Hubka, V. & Eder, W.E.: Theory of Technical Systems, Springer-Verlag. Berlin, 1988.

[5] Andreasen, M.M., Hansen, C.T. & Mortensen, N.H.: The structuring of Products and Product Programmes, Proceeding of the 2’nd WDK Workshop on Product Structuring, June 3-4 1996, Delft University of Technology, Delft, The Netherlands, 1996.

[6] Yu, Bei and Skovgaard, Hans Jørgen, A Configuration Tool to Increase Product competitiveness. IEEE Intelligent Systems and Their applications. July/August 1998.

[7] Haberfellner, R et al: Systems Engineering, Verlag Industrielle Organisation, Zürich, 1995

[8] Klir, J. & Valach, M.: Cybernetic Modelling, Iliffe Books, London, 1965.

[9] Nielsen, M.P. & Harlou, U.: Modelling Product Families In Configuration Systems, M.Sc.-thesis, Department of Control and Engineering Design, Technical University of Denmark, 1999.

[10] Mortensen, N.H.: Design Modelling in a Designer's Workbench - Contribution to a Design Language, Ph.D. thesis, Department of Control and Engineering Design, Technical University of Denmark, Lyngby 1999.

SAT-Based Consistency Checking of Automotive Electronic Product Data

Carsten Sinz and **Andreas Kaiser** and **Wolfgang Küchlin**[1]

Abstract. Complex products such as motor vehicles or computers need to be configured as part of the sales process [3, 8]. If the sale is electronic, then the configuration and some validity checking of the order must be done electronically as part of an electronic product data management system (EPDMS). The EPDMS typically maintains a data base of sales options and parts together with a set of logical constraints expressing valid combinations of sales options and their transformation into manufacturable products. Due to the complexity of these constraints, creation and maintenance of the configuration data base is a nontrivial task and error-prone. We present our system BIS which is commercially used to check global consistency assertions about the product data base used by the EPDMS of a major car and truck manufacturer. The EPDMS uses Boolean logic to encode the constraints, and BIS translates the consistency assertions into problems which it solves using a propositional satisfiability checker. We expect our approach to be especially suited for rapidly changing complex products as they increasingly appear in electronic commerce.

1 Introduction

We present our system BIS, an extension to a commercially used electronic product data management system (EPDMS) at DaimlerChrysler AG, the manufacturer of the Mercedes lines of passenger cars and commercial vehicles. BIS is just now beginning commercial service to weed out residual defects in the product data base that is used by the main EPDMS for order checking and production planning. The EPDMS uses Boolean logic to encode the manufacturability constraints, and BIS translates global consistency assertions on the constraints into problems which it solves using a propositional satisfiability checker. BIS is programmed in modern object-oriented client/server technology and easily runs on a current laptop.

The automotive industry today manages to supply customers with highly individualized products at low prices by configuring each vehicle individually from a very large set of possible options. E.g., the Mercedes C-class of passenger cars knows far more than a thousand options, and on the average more than 30,000 cars will be manufactured before an order is repeated identically. Heavy commercial trucks are even more individualized, and every truck configuration is built only very few times on average.

Traditionally, a sales person will bespeak the individual order with the customer. Still, the space of possible variations is so great that the validity of each order needs to be checked electronically against a product data base which encodes the constraints governing legal combinations of options (e.g., no two radios; no steel sun-roof for convertibles). But the maintenance of a data base with thousands of logical rules is error-prone in itself, especially since it is under constant change due to the phasing in and out of models. Every fault in the data base may lead to a valid order rejected, or an invalid (non constructible) order accepted which may ultimately result in the assembly line to be stopped. Therefore, an EPDMS is employed to maintain the product data base and check orders.

DaimlerChrysler AG, for their Mercedes lines of cars and commercial trucks, employ a mainframe-based EPDMS which does the validity checking of each individual order (followed by initial production planning) in an off-line process. It also supports the on-line maintenance of the product data base by the staff of the product documentation department. The data base contains a large number of constraints formulated in Boolean logic. Some of the constraints represent general rules about valid combinations of sales options, other formulae are attached to parts and express the exact condition under which each part is needed for an order to be manufactured. Based on some years of experience with the current EPDMS, it was found that it is not humanly possible to keep such a large and constantly changing data base of logical rules and formulae absolutely defect-free without help from an automated reasoning system for the documentation logic.

Therefore our system BIS was created as an extension to the current EPDMS to help the product documentation staff in proving complex global consistency assertions about the data base. As an example, BIS can check for each of the thousands of sales options whether it can legally be contained in at least one valid (manufacturable) order. BIS can also deal with partially specified orders, checking e.g. which engine options are still valid given the preselected body and interior, or it can check which parts cannot possibly be part of any vehicles that go to a certain country.

While the current EPDMS is used off-line, it is clear that something similar must be used on-line in an electronic sales system for even simple configurable products, and a system like BIS is then needed to keep the product data base defect free. In an electronic market, there is no more knowledgeable sales-person to assist the customer and to prevent silly orders, and it may not even be feasible to contact the customer personally to sort out problems once an order is submitted. Hence we argue that the methods used in BIS today will be useful in electronic markets with even much simpler products tomorrow.

In Section 2, we shall now summarize, from [7], the capabilities of the core BIS system (BIS 1.0). In Section 3 we shall present some modifications and extensions that went into the current BIS 2.0, based on feedback from actual industrial use of BIS 1.0 on commercial product documentation data.

[1] Symbolic Computation Group, WSI for Computer Science, University of Tübingen and Steinbeis Technology Transfer Center OIT, Sand 13, 72076 Tübingen, Germany, HTTP://WWW-SR.INFORMATIK.UNI-TUEBINGEN.DE

2 BIS: A SAT-Based Consistency Checker for Product Documentation

Before turning to the description of the BIS system, we will need to give a rough picture of the underlying EPDM System which it complements. Then we present some consistency criteria that can be examined using the BIS system, and show how they translate into SAT instances. Thereafter we will shortly comment on the architecture of our system.

2.1 DaimlerChrysler's EPDM System DIALOG

In the following we will describe the EPDM system used for DaimlerChrysler's Mercedes lines more thoroughly.

A customer's order consists of a basic model class selection together with a set of further equipment codes describing additional features. Each equipment code is represented by a Boolean variable, and choosing some piece of equipment is reflected by setting the corresponding variable to *true*. As model classes can be decoded into a set of special equipment codes, all rules in the product documentation are formulated on the basis of codes.

Slightly simplified, each order is processed in three major steps, as is depicted in Figure 1:

1. *Order completion:* Supplement the customer's order by additional (implied) codes.
2. *Constructibility check:* Are all constraints on constructible models fulfilled by this order?
3. *Parts list generation:* Transform the (possibly supplemented) order into a parts list.

All of these steps are controlled by logical rules of the EPDMS. The rules are formulated in pure propositional logic using `AND`, `OR` and `NOT` as connectives, with additional restrictions placed on the rules depending on the processing step, as will be shown below.

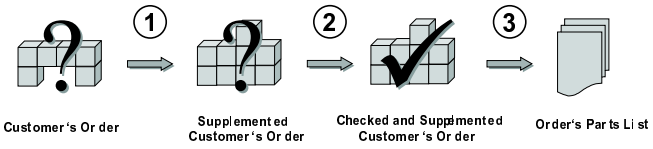


Figure 1. Processing a customer's order.

Order Completion

The order completion (or supplementing) process adds implied codes to an order. The process is guided by special formulae associated with each code, which are of the following form:

$$Cond^S \longrightarrow c,$$

where c is a code (i.e. a propositional variable) and $Cond^S$ an arbitrary formula. The semantics of such a rule is that when condition $Cond^S$ evaluates to *true* under an order, then code c is added to that order. Thus, each rule application extends the order by exactly one code, and the whole completion process is iterated until no further changes result. Ideally, the relationship between original and augmented order should be functional. However, the result of the order completion process may depend critically on the ordering of rule application. We have shown in [7] how to identify potential instances of this problem.

Constructibility Check

In general, constructibility of a customer's order is checked according to the following scheme: For each code, there may be several rules indicating restrictions under which this code may be used. A code is called *constructible* (or *valid*) within a given order if all constraining rules associated with this code are fulfilled, i.e. all of these rules evaluate to *true*. For an order to be *constructible* (or *valid*), each code of the order must be valid. The constructibility check consists of two different parts: The first one is independent of the car model class considered, while the second one takes into account additional features of each car model class. Although the latter incorporates an additional hierarchical organization of rules, we will not elaborate on this. For our purpose the constructibility rules may be considered in a unified, simpler form:

$$c \longrightarrow Cond^C,$$

where c is a code and $Cond^C$ an arbitrary formula. Such a rule expresses the fact that whenever code c occurs in a customer's order, the order must fulfill condition $Cond^C$, i.e. $Cond^C$ must evaluate to *true* for this order.

Parts List Generation

The parts list is subdivided into modules, positions and variants, with decreasing generality from modules to variants. Parts are grouped in modules depending on functional and geometrical aspects. Each position contains all those parts which may be used alternatively in one place. The mutually exclusive parts of a position are specified using variants. Each variant is assigned a formula called a code rule and a part number. A parts list entry is selected for an order if its code rule evaluates to *true*. Thus, to construct the parts list for a completed and checked customer's order, one scans through all modules, positions, and variants, and selects those parts which possess a matching code rule.

2.2 Consistency of Product Documentation

Due to the complexity of the product documentation occurring in the automotive industry, some erroneous rules in the data base are almost unavoidable and usually quite hard to find. Moreover, the rule base changes frequently, and rules often introduce interdependencies between codes which at a first sight seem not to be related at all.

As the rule base not only reflects the knowledge of engineers, but also world wide legal, national and commercial restrictions, the complexity seems to be inherent to automotive product configuration, and is therefore hard to circumvent.

A priori, i.e. without explicit knowledge of intended constraints on constructible models, the following data base consistency criteria may be checked:

Necessary codes: Are there codes which must invariably appear in each constructible order?

Inadmissible codes: Are there any codes which cannot possibly appear in any constructible order?

Consistency of the order completion process: Are there any constructible orders which are invalidated by the supplementing process? Does the outcome of the supplementing process depend on the (probably accidental) ordering in which codes are added?

Superfluous parts: Are there any parts which cannot occur in any constructible order?

Ambiguities in the parts list: Are there any orders for which mutually exclusive parts are simultaneously selected?

Note that the aforementioned criteria are not checked on the basis of existing (or virtual) orders, but constitute intrinsic properties of the product documentation itself.

By incorporating additional knowledge on which car models can be manufactured and which cannot, further checks may be performed. Besides requiring additional knowledge, these tests often do not possess the structural regularity of the abovementioned criteria and thus cannot be handled as systematically as the other tests.

2.3 SAT Encoding of Consistency Assertions

We will now show how to encode the consistency criteria developed in the last section as propositional satisfiability (SAT) problems.

Transformation of the consistency criteria into SAT problems seems to be a natural choice for two reasons: first, the rules of the underlying EPDM system are already presented in Boolean logic; and second, SAT solvers are applied in other areas of artificial intelligence with increasing success [1, 6].

The formulation of all these consistency assertions requires an integrated view of the documentation as a whole or, more precisely, a characterization of the set of orders as they appear having passed the order completion process and the constructibility check. So we first concentrate on a Boolean formula describing all valid, extended orders that may appear just before parts list generation.

Let the set of order completion (supplementing) rules be $SR = \{sr_1, \ldots, sr_n\}$ with $sr_i = Cond_i^S \longrightarrow c_i$. Then the set of completely supplemented orders is described by formula Z, where

$$Z := \bigwedge_{1 \le i \le n} \left(Cond_i^S \Rightarrow c_i \right) .$$

Now, let $CR = \{cr_1, \ldots, cr_m\}$ be the set of constructibility rules with $cr_j = c_j \longrightarrow Cond_j^C$. Then the set of constructible orders is described by formula C, where

$$C := \bigwedge_{1 \le j \le m} \left(c_j \Rightarrow Cond_j^C \right) .$$

Moreover, the set of all orders that have passed the supplementing process and the constructibility control are described by B, where

$$B := Z \wedge C .$$

We now have reached our goal to generate a propositional formula reflecting the state before parts list generation. The mapping of the consistency criteria to SAT instances is now straightforward. For example, code c is inadmissible, iff $B \wedge c$ is unsatisfiable. The other criteria are converted accordingly, but some of them require a more sophisticated translation, especially those tests concerning the order completion process. The complete set of transformations from our consistency assertions to SAT instances – as they are actually built into the BIS system – can be found in [7].

Finally, it should be noted that the process described here is simplified in comparison to the actual order processing that takes place in the DIALOG system. The general ideas should nevertheless be apparent.

2.4 Integration into Work-Flow

We will now briefly describe how the BIS system is integrated into the existing product documentation process.

After having made a change to the documentation rule base (or, alternatively, in regular temporal intervals) some or all of the abovementioned consistency criteria are checked, and potential flaws of the documentation are reported by BIS.

Each inconsistency indicated by BIS has then to be analyzed and interpreted by the product documentation experts: If the product documentation does not correctly reflect reality (in the sense that it does not properly classify what actually can be manufactured), the error has to be corrected – either by adapting the documentation rules or by modifying the product itself. Otherwise the reported inconsistency most likely is an intended exceptional case that does not need any further processing.

Even if not all such inconsistencies are – or even can be – handled, the quality of the product documentation is nevertheless improved. This is an important fact, considering that SAT is an NP-complete problem. Thus, it cannot be guaranteed that the system will find all inconsistencies within a suitably short amount of time. We experienced, however, that for our application worst-case behavior and unacceptably long run-times are the rare exception; the run-times for each proof are usually clearly below one second.

2.5 Architecture of the BIS System

The BIS system is constructed employing object-oriented client/server technology. It consists of a general prover module programmed in C++ with a propositional satisfiability checker as its core component; a C++ server which maintains product data in raw and pre-processed form and handles requests by building the appropriate formulae for the prover; and a graphical user interface programmed in Java, through which tests can be started and results can be displayed. The three components communicate via CORBA interfaces, thereby achieving a great flexibility, allowing e.g. to place each component on a different, suitable computer or to use multiple instances of a component (e.g. prover), if the workload demands this. Figure 2 shows a schematic view of the BIS system architecture.

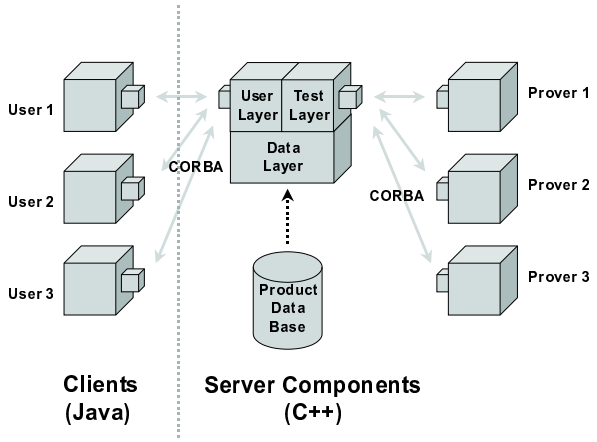


Figure 2. BIS system architecture.

Within the server, the UserLayer is responsible for authentication and handles user requests by starting the appropriate consistency tests. Therefore it employs the TestLayer which in turn is responsible for managing (i.e. scheduling, starting) all consistency checks. The data layer is used as a mediator between the TestLayer and the EPDM system, and supports caching of pre-computed data.

3 Extensions Based on Experience

Since the first evaluation deployment of the BIS 1.0 system (and even before), we have received a lot of feedback from DIALOG users at DaimlerChrysler. This helped us greatly in improving the system in various aspects. We will now describe relevant user feedback as well as experience-induced changes in greater detail.

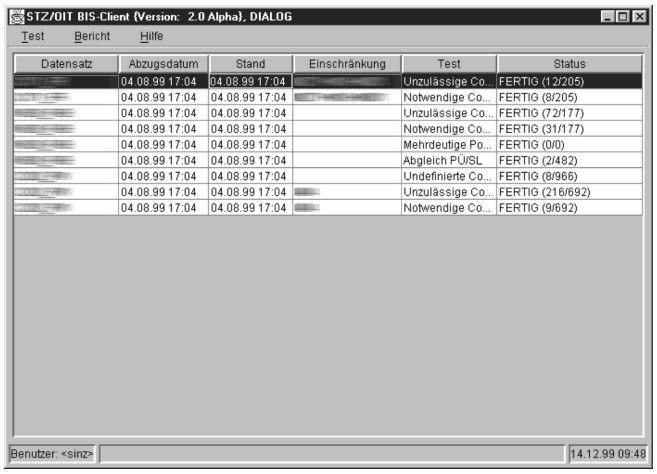


Figure 3. BIS system client.

The following items appeared to be indispensable for a broad everyday use:

Push-button technology: The logical prover component can be completely hidden from the user, and it needs no assistance in finding a proof. Interaction with the prover is done in terms familiar to the operating personnel.

Graphical user interface: The BIS system offers an elaborated graphical user interface, as can be seen from Figure 3. No cryptic command lines have to be typed by the user.

Short response times of the system: As BIS 1.0 was used more and more interactively, consistency checks had to exhibit short and predictable run-times. We will describe below in more detail how we could achieve this.

Customized special checks: Although we offered a general-purpose interface to the prover[2] which could be used to perform a lot of non-standard consistency checks on the product documentation, acceptance of this tool was rather poor. Thus, we implemented a set of further customized special checks and extended the client accordingly.

3.1 Additional Functionality

We will now report on functional extensions that went into BIS 2.0. As we already mentioned above, most of these additional tests could in principle have been performed with the general-purpose test facility of BIS 1.0, but required some kind of – frequently almost trivial – logical problem encoding by the user; an interpretation of the result reported by the client; and often an annoying manual generation of a series of tests.

[2] This interface allowed queries about the existence of valid orders with special properties, where the demanded property is an arbitrary propositional formula.

Restricting the Set of Valid Orders

The formalization mentioned above allows the analysis of the set of all valid orders. However, as it turned out, it is often necessary to restrict the set of orders to be considered to some subset of all valid orders. This may be needed, for example, to check assertions about all valid orders of a certain country, or about all valid orders with a special motor variant.

The formalization of order restrictions could easily be realized by adding a formula R describing the additional restriction to the constructibility formula B, and running the tests on $B \wedge R$.

Valid Additional Equipment Options

Not only for an individual order, but also for a whole class of orders, it may be interesting to know what kind of additional equipment may be selected without making the order invalid. This can be used on the one hand to analyze the product data, but may also serve a customer to find possible extensions of a partially specified order. This can be achieved as follows: Using the restriction possibility of the last paragraph, the partially specified order serves as a restriction R on the set of valid orders to be considered. Then it is checked for all codes c whether or not the formula $B \wedge R \wedge c$ is satisfiable. If this is the case, then code c is a valid extension of the partially specified order.

Combinations of Codes

Upon creation and maintenance of parts list entries, the following question frequently has to be answered: Given a fixed set of codes, which combinations of these codes may possibly occur in a valid order? The answer to this question decides over which parts list entries have to be documented and which have not.

Performing this kind of test one by one for each combination manually appears to be rather cumbersome. An automatic generation of all possible combinations seems to be more appropriate. Again, this can be simply realized by generating systematically all combinations one after the other and checking the corresponding formulae for satisfiability.

Groups of Codes

Although not reflected by the documentation structure, we found it characteristic for automotive product data that certain codes are symmetrically related. We call a set C of codes symmetrically related (with respect to a rule-based product documentation) if there is a non-empty subset R of those rules containing at least one code of set C, such that R is invariant under all permutations of the codes of set C.

A typical case for a set of symmetrically related codes is a set of mutually exclusive codes, where one of the codes must appear in each valid order, i.e. each order must contain exactly one code of the set. For example, in the DIALOG documentation system each order must contain exactly one code that determines the country in which the customer has ordered the car.

Since these kinds of symmetric relations can not be explicitly stated in the EPDM system, but are implicitly given by several rules formed with the binary connectives `NOT`, `AND` and `OR`, we added a possibility to check for a given list of codes (a group of codes) whether or not each valid order contains exactly one of these codes.

This is realized by checking the satisfiability of formula B that describes all valid orders, extended by the additional constraint that the order contains none or at least two of the codes specified in the group.

3.2 Extending the Propositional Language

While sets of mutually exclusive codes represent the most prominent example for a symmetrical relation, one can also think of situations where other symmetrical relations are applicable. For example, a customer can choose exactly one of a set of audio systems or he can completely dispense with audio systems. This means he can choose *at most one* of a set of options. Another example is a valid order that needs exactly k of a set of n colors specified. Obviously giving these kinds of restrictions in standard propositional logic leads to excessive growth in formula size which is certainly unacceptable for user interaction, and may even in the simplest case exhaust the available resources. The fact that the mutual exclusiveness of country codes in DaimlerChrysler's current product documentation is not explicitly stated underlines this.

To address this drawback we added – as is described in detail in [5] – the abovementioned expressions to standard propositional language. This extends the language by expressions of the form $RkX_1, \ldots, X_n$, where $R \in \{=, \neq, \leq, <, \geq, >\}$, k is a positive number and $X_1, \ldots, X_n$ are arbitrary formulae of the extended language. The semantics of such an expression is that exactly Rk of the n formulae are *true*. Thus, the fact that at most one of three possible audio systems A_1, A_2 and A_3 should appear in an order corresponds to the expression

$$\leq 1 : A_1, A_2, A_3 \ ,$$

which is equivalent to writing

$$\neg(A_1 \wedge A_2) \wedge \neg(A_1 \wedge A_3) \wedge \neg(A_2 \wedge A_3)$$

in pure propositional logic.

A closer analysis even shows that any formula in standard propositional logic can be transformed to an equivalent formula based solely on the additional connectives, which differs in size from the original formula only by a constant factor. Thus, these connectives provide us with a method to represent formulae for automotive product data management in a compact, structure-preserving and uniform way.

As a consequence of introducing additional connectives, we refrain from conversion to clausal normal form (CNF) for satisfiability checking – in contrast to most of the commonly used Davis-Putnam-style propositional theorem provers [2]. Although this step involves a more complex prover implementation using a tree data structure (as opposed to integer arrays for CNF representation), its benefit is beyond the mere compactification of formula representation. On formulae generated from automotive product data our prover showed in most cases similar or better performance. Moreover, we avoided an additional data structure to represent the CNF of the formula and therefore could reduce the complexity of the overall system as well as the space requirements and improve response time, because CNF conversion of very large formulae is non-trivial.

Even beyond consistency checking, we consider the introduction of a logical connective that reflects symmetrical relations to be essential to efficiently document product data on the basis of Boolean constraints (see also [9]).

4 Conclusion and Future Work

We presented BIS, a system to complement DaimlerChrysler's automotive EPDM system DIALOG. BIS serves as a tool to increase the quality of the product documentation by allowing to check certain global consistency conditions of the documentation data base as a whole. In BIS 2.0, the consistency assertions and the product documentation rules are translated to formulae of an extended language of propositional logic, which additionally includes a connective for symmetric relations.

Feedback from the documentation personnel showed us which features – among others – should be preferably included into a support tool for product documentation: ease of use via a graphical user interface; good integration into existing work-flow; push-button technology; and short response times.

Although current satisfiability checkers are quite advanced and SAT is still – and increasingly – an area of active research [4], we could learn from the special applicational needs how to improve propositional SAT tools and how to optimize prover techniques. Special constructs occurring frequently in product documentation, such as selection of one out of a set of n entities, are usually not appropriately supported by generic Boolean SAT checkers. Therefore, we see here a wide area of adaptations and improvements on prover technology and possibly further speed gains, brought forward by applicational needs.

For the future, we assume a system like BIS to be indispensable for electronic sales over the World Wide Web. Complex products need to be configured and checked electronically in large numbers, and thus the presence of a correct electronic product documentation receives increased attention. But electronic product documentation shows up to be complex and rapidly changing, thus making maintenance of the product data base difficult. Electronic business may increase the speed of product changes as well as the complexity of products, as product cycles and life times are likely to shrink down. Moreover, the needs of customers may vary considerably around the globe, thus forcing a diversification of the product palette.

Additionally, as we move towards fully electronic sales, configuration and on-line validity checking of even simple products may be required to be aided by electronic support systems in the future, and there will probably be no supporting human sales person. Such systems heavily depend on the quality of electronic product data, which we try to improve with our techniques.

REFERENCES

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, 'Symbolic model checking without BDDs', in *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS. Springer-Verlag, (1999).

[2] M. Davis and H. Putnam, 'A computing procedure for quantification theory', in *Journal of the ACM*, volume 7, pp. 201–215, (1960).

[3] F. Freuder, 'The role of configuration knowledge in the business process', *IEEE Intelligent Systems*, **13**(4), 29–31, (July/August 1998).

[4] I. Gent and T. Walsh, 'Satisfiability in the year 2000', *J. Automated Reasoning*, **24**(1–2), 1–3, (February 2000).

[5] A. Kaiser. Saturation-based satisfiability checking without CNF conversion, 2000. (unpublished manuscript).

[6] H. Kautz and B. Selman, 'Planning as satisfiability', in *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)*, pp. 359–363. John Wiley and Sons, (1992).

[7] W. Küchlin and C. Sinz, 'Proving consistency assertions for automotive product data management', *J. Automated Reasoning*, **24**(1–2), 145–163, (February 2000).

[8] J. McDermott, 'A rule-based configurer of computer systems', *Artificial Intelligence*, **19**(1), 39–88, (1982).

[9] T. Soininen and I. Niemelä, 'Developing a declarative rule language for applications in product configuration', in *Practical Aspects of Declarative Languages, First International Workshop (PADL'99)*, number 1551 in LNCS, pp. 305–319. Springer-Verlag, (1999).

Unified Configuration Knowledge Representation Using Weight Constraint Rules

Timo Soininen[1] and **Ilkka Niemelä**[2] and **Juha Tiihonen**[1] and **Reijo Sulonen**[1]

Abstract. In this paper we present an approach to formally defining different conceptualizations of configuration knowledge, modeling configuration knowledge, and implementing the configuration task on the basis of the formalization. The approach is based on a weight constraint rule language and its efficient implementation designed for representing different aspects of configuration knowledge uniformly and compactly. The use of the language to represent configuration knowledge is demonstrated through formalizing a unified configuration ontology. The language allows a compact and simple formalization. The complexity of the configuration task defined by the formalization is shown to be **NP**-complete.

1 Introduction

Product configuration can be roughly defined as the task of producing a specification of a product individual, a *configuration*, from a set of predefined component types while taking into account a set of restrictions on combining the component types. Knowledge based systems for configuration tasks, *product configurators*, have recently become an important application of artificial intelligence techniques for companies selling products tailored to customer needs.

Several approaches to configuration are based on relatively well-understood general formalisms, such as constraint satisfaction, its extensions, and variants of description logics. Other approaches have defined specific configuration domain oriented conceptual foundations. These include the three main conceptualizations of configuration knowledge as resource balancing, product structure and connections within a product [11]. Despite the research, there are few formal models of configuration aiming to unify the different formal and conceptual approaches. Such models are needed to facilitate rigorous analysis and comparison of the different approaches.

In this paper we present an approach that facilitates formal representation of configuration knowledge based on different and unified conceptualizations. In contrast to many previous approaches that have developed special purpose algorithms for each conceptualization, our aim is to provide for fast prototyping of conceptualizations and configuration tasks based on them. This is accomplished by capturing the configuration knowledge using a neutral representation language called weight constraint rules and implementing the configuration task using a general implementation of such rules. Therefore, when changing the configuration model or its underlying conceptual foundation, only the representation of the knowledge is changed. There is no need to design a new algorithm for the configuration task, as the general implementation of the rule language can still be used.

We first briefly introduce the logic program like rule language that has been specifically designed to allow representing knowledge on different aspects of configuration in a uniform and simple manner. At the same time, the computational efficiency of the configuration task has also been taken into account. The language has a declarative semantics providing a formal definition for the product configuration task. The semantics captures the property that a configuration can only contain elements that are *justified* by the configuration model. This property has been identified as important in e.g. the research on dynamic constraint satisfaction problems (DCSP) [5]. The justification property also allows a more compact and modular representation of knowledge than e.g. classical logic or constraint satisfaction problems [10, 9]. For example, a subset of the rule language can represent e.g. CSP and DCSP in a simple, compact manner [10]. It also extends these approaches with *cardinality* and *weight constraints*.

After introducing the rule language we demonstrate its applicability to configuration knowledge representation. This is done by showing how a configuration model represented using a simplification of a generalized configuration ontology [11] covering product structures, resource balancing, connections and constraints is mapped to a set of rules. The rules are not used by product modelers but as an intermediate language that a higher level configuration modeling language is translated to. Due to the built-in justification property of the language, the mapping is modular and relatively simple. Extending the representation to cover a more complex conceptualization is straightforward, although we discuss in brief some challenges to this. The relevant decision problem for the configuration task is defined and shown to be **NP**-complete on the basis of the mapping to rules. Finally, we discuss our approach in relation to previous similar work.

2 Weight Constraint Rules

In this section we briefly introduce the weight constraint rule language used for formalizing configuration knowledge (for more information on the language see [7, 8]). The language extends logic programs by offering support for expressing choices as well as cardinality and resource constraints that are often useful in configuration:

Example 1. A (partial) configuration model of a simplified PC could consist of the following knowledge: There is a known set of different types of IDE hard disks, IDE CD ROMs, scanners and SCSI cards that can be parts of a PC. The different IDE hard disks and IDE CD ROMs are IDE devices. A PC must have from one to two IDE devices, of which at least one must be an IDE hard disk. In addition, a desktop publishing PC must have a scanner, which can be a flatbed or a slide scanner. As the slide scanner is a SCSI device, including it in

[1] Helsinki University of Technology, TAI Research Center and Lab. of Information Processing Science, P.O.Box 9555, FIN-02015 HUT, Finland, email: {Timo.Soininen,Juha.Tiihonen,Reijo.Sulonen}@hut.fi

[2] Helsinki University of Technology, Dept. of Computer Science and Eng., Lab. for Theoretical Computer Science, P.O.Box 5400, FIN-02015 HUT, Finland, email: Ilkka.Niemela@hut.fi

a configuration requires a SCSI card. There are two different types of SCSI cards to select from. Furthermore, each component has an associated price. The customer might require that the total price, summed over the components in a configuration, must be less than $900.

The rule language aims to capture the type of configuration knowledge in the example in a compact form. To do this, it is equipped with *conditional literals* restricted by *domain predicates* for representing the different types of components and other objects and the knowledge on them, such as the IDE devices above. *Cardinality constraints* are introduced for representing choices with lower and upper bounds, such as for the IDE devices in the above example. Resource constraints on a configuration exemplified by the prices of the components above are captured in the generalized language using *weight constraints*, which give the language its name.

First we consider ground rules, i.e., rules where variables quantifying over a rule are not allowed. Then we introduce rules with variables. A *weight constraint rule*

$$C_0 \leftarrow C_1, \ldots, C_n \tag{1}$$

is built from *weight constraints* C_i of form

$$L \quad \leq \quad \{a_1 = w_1, \ldots, a_n = w_n, \\ \text{not } a_{n+1} = w_{n+1}, \ldots, \text{not } a_m = w_m\} \leq U \tag{2}$$

where L and U are two numbers giving the lower and upper bound for the constraint, each a_i is an atomic formula and each w_i a number giving the weight for the corresponding literal (an atom or its negation). Such a rule is intuitively read as follows: if the *body*, i.e. each of C_1 to C_n, holds then the *head* C_0 must also hold. The semantics for such rules is given in terms of models that are sets of ground atoms. We say that a positive literal a is satisfied by a model S if $a \in S$ and a negative literal $\text{not } a$ is satisfied if $a \notin S$. A weight constraint is satisfied by a model S if the sum of weights of the literals satisfied by S is between the bounds L and U. Note that "$<$" could also be similarly used in a constraint. E.g.,

$$30 \leq \{ \ part(pc_1, d_1) = 10, part(pc_1, d_2) = 20, \\ part(pc_1, d_3) = 30, part(pc_1, d_4) = 30 \ \} \leq 40$$

is satisfied by a model $\{part(pc_1, d_1), part(pc_1, d_2)\}$ for which the sum of the weights is $10 + 20 = 30$ but not by a model $\{part(pc_1, d_3), part(pc_1, d_4)\}$ for which the sum is 60.

We use shorthands for some special cases of weight constraints. A *cardinality constraint* where all weights are 1 is written as

$$L \ \{a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_m\} \ U \tag{3}$$

and a cardinality constraint $1\{l\}1$ with one literal and 1 as both bounds simply as a literal l. We call rules where all constraints C_i are literals *normal rules*. We can also write mixed rules such as

$$1 \ \{part(pc_1, sc_1), part(pc_1, sc_2)\} \ 1 \leftarrow part(pc_1, sscan) \tag{4}$$

The semantics of a set of rules is captured by a subclass of models called *stable models*. They fulfill two important properties: (i) a stable model *satisfies* the rules and (ii) is *justified* by the rules. A rule r of form (1) is satisfied by a model S iff S satisfies C_0 at least whenever it satisfies each of $C_1, \ldots, C_n$. A rule without a head is satisfied if at least one of the body constraints is not. The technical details of the justifiability property can be found in [7], but we explain the basic intuition through an example. Consider rule (4). It is satisfied, e.g., by the model $\{part(pc_1, sc_1)\}$. However, this model is not justified by the rule since there is no reason to include the head of the rule in the model, since the body of the rule, $part(pc_1, sscan)$, is not justified. Indeed, for the rule (4) the only stable model is the empty set which also satisfies the rule. Suppose we add two rules

$$1 \ \{part(pc_1, fscan), part(pc_1, sscan)\} \ 2 \quad \leftarrow \quad dtp(pc_1) \tag{5}$$

$$dtp(pc_1) \quad \leftarrow \tag{6}$$

to (4). Then each stable model of the rules (4–6) contains the atom $dtp(pc_1)$ and at least one of $\{part(pc_1, fscan), part(pc_1, sscan)\}$. This holds because $dtp(pc_1)$ is justified by rule (6) and, hence, a choice from $\{part(pc_1, fscan), part(pc_1, sscan)\}$ is justified by (5). If $part(pc_1, sscan)$ is taken, then by (4) the choice between $\{part(pc_1, sc_1), part(pc_1, sc_2)\}$ is justified. So this implies that $\{dtp(pc_1), part(pc_1, fscan), part(pc_1, sscan), part(pc_1, sc_1)\}$ is one stable model for the rules (4–6) but this not the case for the set $\{dtp(pc_1), part(pc_1, fscan), part(pc_1, sc_1)\}$ as $part(pc_1, sc_1)$ is not justified. We note that there is an important difference between a rule with and a rule without a head: only the former can bring something into a stable model. The latter only exclude stable models.

For applications it is very useful to provide rules where variables quantifying over the whole rule can be used. With the help of variables one can write rules in a compact and structured way and this facilitates developing, updating and maintaining a rule set.

The semantics for rules with variables [7] is obtained by considering the ground instantiation of the rules with respect to their Herbrand universe, i.e. all variable-free (ground) terms that can be built from the constants and functions in the rules. However, in this general case the rule language is highly undecidable. A decidable subclass of rules with variables is obtained by considering the function-free and *domain-restricted* case [7]. Here the intuition is that each variable has a domain predicate which provides the domain over which the variable ranges. Domain predicates can be defined using normal rules starting from basic facts. Hence, it is possible to define new domain predicates from already defined ones using union, intersection, complement, join, and projection. As an example, consider two sets of ground facts $\{ide_{hd}(h_i) \leftarrow\}$ and $\{ide_{cd}(c_i) \leftarrow\}$ giving the available IDE hard disks and IDE CD ROMs, respectively. Now both ide_{hd} and ide_{cd} can be used as domain predicates. We can also define a new domain predicate ide as their union using the two rules

$$ide(X) \quad \leftarrow ide_{hd}(X) \quad ; \quad ide(X) \quad \leftarrow ide_{cd}(X)$$

where the variable X quantifies universally over a rule. Note that we use ";" as a separator of rules. We use the convention that variables start with a capital and constants with a lower case letter. A rule is domain-restricted if every variable in it appears in a domain predicate in the body of the rule. E.g., the rule

$$1 \ \{part(X, d_1), \ldots, part(X, d_m)\} \ 2 \leftarrow pc(X) \tag{7}$$

is domain-restricted if $pc(X)$ is a domain predicate. A simple way of understanding the semantics of the rules with variables is to see them as shorthands for sets of ground rules. For example, rule (7) can be understood as a set of ground rules of form

$$1 \ \{part(pc_i, d_1), \ldots, part(pc_i, d_m)\} \ 2 \leftarrow pc(pc_i) \tag{8}$$

where X is replaced by each constant pc_i for which $pc(pc_i)$ holds.

In order to compactly represent sets of literals in constraints *conditional literals*, i.e., expressions such as $part(X, D) : ide(D)$ can be used in place of literals. Thus, rule (7) can be expressed as

$$1 \ \{part(X, D) : ide(D)\} \ 2 \leftarrow pc(X) \tag{9}$$

A rule with conditional literals is *domain-restricted* if each variable in it appears in a positive domain predicate in body or the conditional part (like $ide(D)$ above) of some conditional literal in the rule.

When using conditional literals we need to distinguish between *local* and *global* variables in a rule. The idea is that global variables quantify universally over the whole rule but the scope of a local variable is a single conditional literal. We do not introduce any notation to make the distinction explicit but use the following convention: a variable is local to a conditional literal if it appears only in this literal in the rule and all other variables are global to the rule. For example,

in the rule (9) the variable D is local to the conditional literal in the head but X is a global variable. Again a rule with conditional literals can be understood as a shorthand for a set of ground rules resulting from a two step process. First all global variables are replaced by all ground terms satisfying the domain predicates in every possible way. This gives a set of rules where variables appear only as local variables in conditional literals. For example, for the rule (9) elimination of the global variable X leads to a set of rules of the form

$$1\ \{part(pc_i, D) : ide(D)\}\ 2 \leftarrow pc(pc_i) \tag{10}$$

one for each ground term pc_i for which $pc(pc_i)$ holds. In the second step, for each such rule, local variables are eliminated by replacing each conditional literal by a sequence of ground instances of the main predicate such that domain predicate in the conditional part holds. For example, in rule (10) the conditional literal $part(pc_i, D)\ :\ ide(D)$ is replaced by a sequence

$$part(pc_i, d_1), \ldots, part(pc_i, d_m)$$

where the ground terms $d_1, \ldots, d_m$ are the only terms for which $ide(d_i)$ holds. Hence, the rule (9) can been seen as a shorthand for a set of ground rules of the form (8). Conjunctions of form

$$part(P, D) : pc(P) : ide(D)$$

are also allowed in conditional parts, corresponding to a sequence of ground facts $part(pc_i, d_j)$ for which both $pc(pc_i)$ and $ide(d_j)$ hold.

Domain predicates such as $cost(X, C)$ giving the unique cost C of each component X can also be used for defining weights:

$$\leftarrow 900 \leq \{part(X, Y) : cost(Y, C) = C\},\ \ pc(X) \tag{11}$$

is a shorthand for a set of ground rules containing a rule

$$\leftarrow 900 \leq \{part(pc_i, d_1) = c_1, \ldots, part(pc_i, d_n) = c_n\},\ pc(pc_i)$$

for each pc_i s.t. $pc(pc_i)$ holds and where $(d_1, c_1), \ldots, (d_n, c_n)$ are the only pairs for which the domain predicate $cost(d_i, c_i)$ holds.

Example 2. Consider Example 1. Using the rule language the configuration model can be represented in the following way. Basic component types are given as corresponding facts and normal rules defining domain predicates as for ide. The rule (9) captures the requirement that a PC has from one to two IDE devices and a rule

$$\leftarrow \{part(X, D) : ide_{hd}(D)\}\ 0,\ \ pc(X)$$

the property that at least one of them is an IDE hard disk. The requirements for the scanners and SCSI cards are captured using rules (4–5) and the constraint on the total price of the components can be stated using a rule like (11).

An implementation of the weight constraint rule language called **Smodels** is publicly available at `http://www.tcs.hut.fi/Software/smodels/`. It computes stable models of domain-restricted rules using a precompilation technique and a Davis-Putnam like procedure which employs efficient search space pruning techniques and a powerful dynamic application-independent search heuristic. The current implementation supports only non-recursive definitions of domain predicates for efficiency, and integer weights in order to avoid complications due to finite precision of standard real number arithmetic. **Smodels** is competitive even against special purpose systems, e.g., in planning and satisfiability problems [6].

3 Configuration Knowledge

In this section we show how to represent configuration knowledge using the rule language. We distinguish between the rules giving *ontological definitions* and the rules representing a configuration model. The former are not changed when defining a new configuration model and are enclosed in a box in the following. The latter appear only in the examples. We further make the convention that the domain predicates are typeset normally whereas other predicates defining the configuration are typeset in $\boldsymbol{boldface}$.

The representation is based on a simplified version of a general configuration ontology [11]. In the ontology, there are three main categories of knowledge: *configuration model knowledge*, *requirements knowledge* and *configuration solution knowledge*. Configuration model knowledge specifies the entities that can appear in a configuration and the rules on how the entities can be combined. In our approach, it is represented as a set of rules. Configuration solution knowledge specifies a configuration, defined in our approach as a stable model of the set of rules in the configuration model. Requirements knowledge specifies the requirements on a configuration and is defined as a set of rules that a configuration must satisfy.

3.1 Types, Individuals and Taxonomy

Most approaches to configuration distinguish between *types* and *individuals*, often called classes and instances. Types in a configuration model define intensionally the properties, such as parts, of their individuals that can appear in a configuration. In the simplified ontology, a configuration model includes the following disjoint sets: *component types*, *resource types*, *port types* and *constraints*. The types are organized in a *taxonomy* or *class hierarchy* where a subtype *inherits* the properties of its supertypes in the usual manner. For simplicity we only allow individuals of concrete, i.e. leaf, types of the taxonomy, which unambiguously describe the product, in a configuration.

Individuals of concrete port and component types are naturally represented as object constants with unique names. This allows several individuals of the same type in a configuration. Types are represented by unary domain predicates ranging over their individuals. Since a resource of a given type need not be distinguished as an individual, there is exactly one individual of each concrete resource type. The individuals that are included in a configuration are represented by the unary predicate $\boldsymbol{in}()$ ranging over individuals.

The type predicates are used as the conditional parts of literals to restrict the applicability of the rules to individuals of the type only. This facilitates defining properties of individuals (see below). The type hierarchy is represented using rules stating that the individuals of the subtype are also individuals of the supertype. This effects monotonic inheritance of the property definitions.

Example 3. A computer is represented by component type pc (PC), which is a subtype of cmp, the generic component type. For each individual pc_i of component type pc in a configuration it holds that $pc(pc_i)$. Component type ide (IDE device) is also a subtype of cmp. A type representing IDE hard disks, hd, is a subtype of ide. Actual hard disks are represented as subtypes of hd, namely hd_a and hd_b. IDE CD ROM devices cd_a and cd_b are subtypes of type cd, which is a subtype of ide. Software packages are represented by type sw. Software package types sw_a and sw_b are subtypes of sw. Port and resource types are introduced in the following sections. The following rules define the component types and their taxonomy:

$$\begin{array}{lll} cmp(C) \leftarrow pc(C) & ; ide(C) \leftarrow hd(C) & ; hd(C) \leftarrow hd_a(C) \\ cmp(C) \leftarrow ide(C) & ; ide(C) \leftarrow cd(C) & ; hd(C) \leftarrow hd_b(C) \\ cmp(C) \leftarrow sw(C) & ; sw(C) \leftarrow sw_a(C) & ; cd(C) \leftarrow cd_a(C) \\ & sw(C) \leftarrow sw_b(C) & ; cd(C) \leftarrow cd_b(C) \end{array}$$

3.2 Compositional Structure

The decomposition of a product to its parts, referred to as *compositional structure*, is an important part of configuration knowledge. A component type defines its direct parts through a set of *part definitions*. A part definition specifies a *part name*, a *set of possible part types* and a *cardinality*. The part name identifies the role in which a component individual is a part of another. The possible part types indicate the component types whose component individuals are allowed in the part role. The cardinality, an integer range, defines how many component individuals must occur as parts in the part role.

For simplicity, we assume that there is exactly one independent component type, referred to as *root component type*. An individual of this type serves as the root of the product structure. In a configuration there is exactly one individual of the root type. Component types are also for simplicity assumed to be exclusive meaning that a component individual is part of at most one component individual. Further, no component type can have itself or any of its super- or subtypes as a possible part type in any of its part definitions or the part definitions of its possible part types, and so on recursively. This implies that a component individual is not directly or transitively a part of itself. These restrictions are placed to prevent counterintuitive structures of physical products. In effect the compositional structure of a configuration forms a tree of component individuals, and each component individual in a configuration is in the tree.

The fact that a component individual has as a part another component individual with a given part name is represented by the tertiary predicate $\boldsymbol{pa}()$ on the whole component individual, the part component individual and the part name. A part name is represented as an object constant and the set of part names in a configuration model are captured using the domain predicate $pan()$.

A part definition is represented as a rule that employs a cardinality constraint in the head. The individuals of possible part types in a given part definition of a given component type are represented using a domain predicate ppa. It is defined as the union of the individuals of the possible component types.

Example 4. The root component type pc has as its parts 1 to 2 mass-storage units (with part name ms) of type ide, and 0 to 10 optional software packages (with part name swp) of type sw. Note that using an abstract (non-leaf) type, such as ide, in a part definition effectively enables a choice from its concrete subtypes. The fact that pc is the root component type and the part names and possible part types of PC are represented as follows:

$$root(C) \leftarrow pc(C) \quad ; \quad ppa(C_1, C_2, ms) \leftarrow pc(C_1), ide(C_2)$$
$$pan(ms) \leftarrow \quad ; \quad ppa(C_1, C_2, swp) \leftarrow pc(C_1), sw(C_2)$$
$$pan(swp) \leftarrow$$

The part definitions for the mass storage and software package roles are represented as follows:

$$1\ \{\boldsymbol{pa}(C_1, C_2, ms) : ppa(C_1, C_2, ms)\}\ 2 \leftarrow \boldsymbol{in}(C_1), pc(C_1)$$
$$0\ \{\boldsymbol{pa}(C_1, C_2, swp) : ppa(C_1, C_2, swp)\}\ 10 \leftarrow \boldsymbol{in}(C_1), pc(C_1)$$

The ontological definitions that exactly one individual of the root type is in a configuration, and that other component individuals are in a configuration if they are parts of something are given as follows:

$$1\ \{\boldsymbol{in}(C) : root(C)\}\ 1 \leftarrow$$
$$\boldsymbol{in}(C_2) \leftarrow \boldsymbol{pa}(C_1, C_2, N), cmp(C_1), cmp(C_2), pan(N)$$

The exclusivity of component individuals is captured by the following ontological definition that a component individual can not be a part of more than one component individual:

$$\leftarrow cmp(C_2), 2\ \{\boldsymbol{pa}(C_1, C_2, N) : cmp(C_1) : pan(N)\}$$

3.3 Resources

The resource concept is useful in configuration for modeling the production and use of some more or less abstract entity, such as power or disk space. Some component individuals produce resources and other component individuals use them. The amount produced must be greater than or equal to the amount used.

A component type specifies the resource types and amounts its individuals produce and use by *production definitions* and *use definitions*. Each production or use definition specifies a *resource type* and a *magnitude*. The magnitude specifies how much of the resource type component individuals produce or use.

A resource type is represented as a domain predicate. Only one resource individual with the same name as the type is needed, since a resource is not a countable entity. A production and a use definition of a component type is represented using a tertiary domain predicate $prd()$ on component individuals of the producing or using component type, individual of the produced or used resource type and the magnitude. Use is represented as negative magnitude.

Example 5. Disk space is used by the software packages and produced by hard disks. Disk space is represented by resource type ds. Each subtype of type sw uses a fixed amount of disk space, represented by their use definitions: sw_a uses 400MB and sw_b 600 MB. Hard disks of type hd_a produce 700MB and of type hd_b 1500MB of ds. The following rules represent the ds resource type and the production and use definition of component types:

$$prd(C, ds, -400) \leftarrow sw_a(C); \qquad res(R) \leftarrow ds(R)$$
$$prd(C, ds, -600) \leftarrow sw_b(C); \qquad ds(ds) \leftarrow$$
$$prd(C, ds, 1500) \leftarrow hd_b(C); \qquad prd(C, ds, 700) \leftarrow hd_a(C)$$

The production and use of a resource type by the component individuals is represented as weights of the predicate $\boldsymbol{in}()$. The ontological definition that the resource use must be satisfied by the production is expressed with a weight constraint rule stating that the sum of the produced and used amounts must be greater than or equal to zero:

$$\leftarrow \quad res(R), \{\boldsymbol{in}(C) : prd(C, R, M) = M\} < 0$$

3.4 Ports and Connections

In addition to hierarchical decomposition, it is often necessary to model connections or compatibility between component individuals. A *port type* is a definition of a connection interface. A *port individual* represents a "place" in a component individual where at most one other port individual can be connected. A port type has a *compatibility definition* that defines a set of port types whose port individuals can be connected to port individuals of the port type.

A component type specifies its connection possibilities by *port definitions*. A port definition specifies a *port name*, a port type and *connection constraints*. Port individuals of the same component individual cannot be connected to each other. For simplicity, we consider only a limited connection constraint specifying whether a connection to a port individual is obligatory or optional. Effectively an obligatory connection sets a requirement for the existence of and connection with a port of a compatible component individual.

Port types are represented as domain predicates and port individuals as uniquely named object constants. Compatibility of port types is represented as the binary domain predicate $cmb()$ on port individuals of compatible port types and a rule that any two compatible port individuals can be connected. The connections are represented as the symmetric, irreflexive binary predicate $\boldsymbol{cn}()$ on two port individuals.

A port individual is connected to at most one other port individual. The following rules represent these ontological definitions:

$$
\begin{array}{rcl}
0\ \{\boldsymbol{cn}(P_1,P_2)\}\ 1 & \leftarrow & \boldsymbol{in}(P_1), \boldsymbol{in}(P_2), cmb(P_1,P_2) \\
\boldsymbol{cn}(P_2,P_1) & \leftarrow & \boldsymbol{cn}(P_1,P_2), prt(P_1), prt(P_2) \\
& \leftarrow & prt(P_1), 2\ \{\boldsymbol{cn}(P_1,P_2) : prt(P_2)\} \\
& \leftarrow & prt(P_1), \boldsymbol{cn}(P_1,P_1)
\end{array}
$$

Example 6. The configuration model includes port types ide_c and ide_d that are subtypes of the general port type prt. These types represent the computer and peripheral device sides of IDE connection. The compatibility definition of ide_c states that it is compatible with ide_d. Correspondingly ide_d states compatibility with ide_c. The following rules represent the port types and compatibility definitions:

$$
\begin{array}{lcl}
prt(P) \leftarrow ide_c(P) & ; & cmb(P_1,P_2) \leftarrow ide_c(P_1), ide_d(P_2) \\
prt(P) \leftarrow ide_d(P) & ; & cmb(P_1,P_2) \leftarrow ide_d(P_1), ide_c(P_2)
\end{array}
$$

A port definition of a component type is represented as a rule very similar to a part definition, but with the tertiary predicate po signifying that a component individual has a port individual with a given port name. The pon predicate captures the port names.

Example 7. Component type pc has two ports with names ide_1 and ide_2 of type ide_c for connecting IDE devices. Component type ide has one port of type ide_d, called ide_p, for connecting the device to a computer. The ide_p port has a connection constraint that connection to that port is obligatory. In rule form:

$$
\begin{array}{l}
pon(ide_1) \leftarrow \ ; \ pon(ide_2) \leftarrow \ ; \ pon(ide_p) \leftarrow \\
1\ \{\boldsymbol{po}(C,P,ide_1) : ide_c(P)\}\ 1 \quad \leftarrow \boldsymbol{in}(C), pc(C) \\
1\ \{\boldsymbol{po}(C,P,ide_2) : ide_c(P)\}\ 1 \quad \leftarrow \boldsymbol{in}(C), pc(C) \\
1\ \{\boldsymbol{po}(C,P,ide_p) : ide_d(P)\}\ 1 \quad \leftarrow \boldsymbol{in}(C), ide(C) \\
\leftarrow ide(C), ide_d(P_1), \boldsymbol{po}(C,P_1,ide_p),\ \{\boldsymbol{cn}(P_1,P_2) : prt(P_2)\}\ 0
\end{array}
$$

The ontological definitions that a port individual is in a configuration if some component individual has it and that port individuals of one component individual cannot be connected are also needed:

$$
\begin{array}{rcl}
\boldsymbol{in}(P) & \leftarrow & cmp(C), pon(N), prt(P), \boldsymbol{po}(C,P,N) \\
& \leftarrow & cmp(C), pon(N_1), prt(P_1), \boldsymbol{po}(C,P_1,N_1), \\
& & pon(N_2), prt(P_2), \boldsymbol{po}(C,P_2,N_2), \boldsymbol{cn}(P_1,P_2)
\end{array}
$$

3.5 Constraints

All approaches to configuration have some kinds of *constraints* as a mechanism for defining the conditions that a correct configuration must satisfy. Rules without heads are used for this in our approach.

Example 8. In the PC configuration model, there is a constraint that a hard disk of type hd must be part of PC:

$$\leftarrow pc(PC_1),\ \{\boldsymbol{pa}(C_1,C_2,N) : hd(C_2) : pan(N)\}\ 0$$

4 Computational Complexity

We make the following assumptions. A configuration model $\mathcal{CM}$ represented according to the ontology is translated to a set of rules CM as in Section 3, including the rules for ontological definitions. Then, a set of ground facts S is added, providing the individuals that can be in a configuration. S is constructed out of the domain predicates representing the concrete types in CM and unique object constants for the individuals of concrete types. The set of rules $CM \cup S$ is all that is needed for representing the product, and subsequently configuring it. The requirements are represented using another set of rules R.

The set S can be thought of as a storage of individuals from which a configuration is to be constructed. The set S thus induces an upper bound on the size of a configuration. This is important since if such a bound cannot be given, the configurations could in principle be arbitrarily large, even infinite. However, for any configuration model $\mathcal{CM}$ agreeing with the ontology in Section 3, a finite, bounded set $max_i(\mathcal{CM})$ containing the maximum number of individuals of any concrete type can be shown to exist. It can be constructed using the following observations. For each concrete resource type there is exactly one individual. Every concrete component individual is a node in the maximal compositional structure tree rooted at the unique root component individual. The tree can be constructed by going through the part definitions starting from the root component type. Finally, the number of port individuals is bounded by the number of component individuals. Now, a set S defining enough individuals for any correct configuration w.r.t. $\mathcal{CM}$ can be constructed by adding a fact $t(i_j) \leftarrow$ for each individual i_j in $max_i(\mathcal{CM})$ whose concrete type is t.

We further make the assumption that the number of variables in the rule representation of each constraint in $\mathcal{CM}$ and each rule in R is bounded by some constant c_l. This is based on the observation that even checking whether a constraint rule or requirement rule of arbitrary length is satisfied by a configuration is computationally very hard. This would be contrary to the intuition that checking whether a constraint or requirement is in effect in a configuration should be easy. Since the ontological definitions in CM have at most five variables, this assumption implies that there is a bound $c = \max(c_l, 5)$ on the number of variables in any rule of any CM and R.

The above assumptions lead to the following definition of the decision version of the configuration task:

Definition 9. CONFIGURATION TASK(D): Given a configuration model $\mathcal{CM}$ translated to a set of rules CM, a set of ground facts S, and a set of rules R, where the number of variables in any rule of CM and R is bounded by a constant c, is there a configuration C, a stable model of $CM \cup S$, such that C satisfies R?

Theorem 4.1. *CONFIGURATION TASK(D) is* ***NP****-complete in the size of* $CM \cup S \cup R$.

Proof. (Sketch) Task is in **NP**: For CONFIGURATION TASK(D) there is the following polynomial time non-deterministic algorithm:

1. Guess a configuration C. It holds that C is of polynomial size w.r.t. $|CM \cup S \cup R|$ since C is a subset of the Herbrand Base (H_B) of $CM \cup S$ and $|H_B|$ is at most polynomial w.r.t. $|CM \cup S|$. The latter holds since the number of variables in any rule of $CM \cup S$ is bounded by c and the size of the Herbrand Universe of CM is bounded by $|CM \cup S|$.
2. Check that C is a stable model of $CM \cup S$. This is accomplished by instantiating $CM \cup S$ with its Herbrand Universe, thus obtaining the set of ground rules $(CM \cup S)_G$, and checking that C is a stable model of $(CM \cup S)_G$. Since the number of variables in any rule of $CM \cup S$ is bounded by c and the size of the Herbrand Universe of CM is bounded by $|CM \cup S|$, $|(CM \cup S)_G|$ is polynomial w.r.t. $|CM \cup S|$ and the instantiation can also be computed in polynomial time. Checking if C is a stable model of a set of instantiated rules can be done in polynomial time [7].
3. Check that C satisfies R. This can be accomplished in polynomial time similarly as in Step 2 by instantiating $R \cup S$ with its Herbrand Universe and checking that C satisfies $(R \cup S)_G$.

NP-hardness: The **NP**-complete 3-SAT problem of whether a propositional sentence $(l_1 \vee l_2 \vee l_3) \wedge \ldots \wedge (l_{m-2} \vee l_{m-1} \vee l_m)$

is satisfiable can be reduced to the configuration task e.g. as follows: Introduce in $\mathcal{CM}$ a root component type r and add a distinct component type for each variable that appears in the sentence. Add to r a part definition for each variable such that the component type is the same as the variable, and the cardinality is $[0, 1]$. For each clause $(l_1 \vee l_2 \vee l_3)$, introduce the constraint rule $\leftarrow\ t(l_1), t(l_2), t(l_3)$, where $t(l_i) = \text{not}\ \ in(X), v_i(X)$ if l_i is a positive literal and $t(l_i) = in(X), v_i(X)$ if l_i is a negative literal, where v_i is the variable in l_i. Finally, construct S by including a fact $r(r_1) \leftarrow$ and the facts $v_i(v_{i1}) \leftarrow$ for each variable v_i. Now, the sentence is satisfiable iff there is a configuration of CM and S satisfying $R = \emptyset$.

□

5 Previous Work

There are several approaches that define a configuration oriented language, a mapping to an underlying language, and implement the configuration task using an implementation of the underlying language. Our approach differs from these in the following respects. The underlying languages do not always have a clear declarative semantics or their implementations are incomplete. They were usually not designed for configuration typical knowledge representation. Finally, the complexity of the configuration task has not been precisely classified. The last issue holds also for the approaches discussed next.

Some exceptions to the above pattern exist. In [1], a formal definition of configuration as constructing a Herbrand model of a description logic-like language is given. Another approach defining (implicitly) configurations as models of a language is described in [4], where a component and connection based ontology is formalized using Ontolingua. In [1], the ontological foundation of configuration is not explored, whereas in [4] the ontology is more restricted than ours.

The approach with perhaps the most similar goals to ours is described in [2, 3]. A similar configuration domain oriented conceptualization and a configuration model are given a formal semantics using a restricted variant of predicate logic extended with set constructs [2]. This definition is then mapped to a generative constraint satisfaction problem (GCSP), and implemented using its implementation. In our approach, the formalization is done directly using the language that provides the implementation. In contrast to our approach, the configuration task is cast as the task of finding a subset minimal set of sentences representing the configuration such that the union of the configuration model and the configuration together with requirements is consistent. The subset minimality condition is a stronger form of justification than ours based on a fixpoint definition [7].

The mapping from the conceptualization to the formalization in [2] is more complex than in our approach. This is partially due to a more complex conceptualization, but there are also other significant differences. First, unlike in [2], our mapping is modular in the sense that each type, property definition and constraint can be formalized independently of other things in the configuration model. The structure of the product is represented in [2] using ports and connections, which leads to more complexity. Furthermore, defining the configuration as a stable Herbrand model eliminates the need for "closure" axioms required in [3, 2] to restrict the set of sentences corresponding to a configuration to be complete and not to contain extraneous things. In addition, the weight constraints allow a compact representation of things which require complex predicates in [2].

It is relatively easy to extend our representation to cover a more complex conceptualization such as those in [11, 2]. These include additional concepts for representing attributes and functionality and additional definitions for the concepts. However, some aspects cannot be captured in a straightforward manner. E.g., the implementation does not yet include arithmetic for reals which is useful for expressing, e.g., attribute values and resource amounts. However, integer arithmetic is included as built-in functions.

6 Conclusions and Future Work

We have presented an approach to formally defining different conceptualizations of configuration knowledge, modeling configuration knowledge, and implementing the configuration task on the basis of the formalization. The approach is based on a novel weight constraint rule language designed for representing different aspects of configuration knowledge uniformly and in a straightforward manner. Using the language to represent different aspects of configuration knowledge is demonstrated through formalizing a unified configuration ontology. The language allows a compact and simple formalization. The complexity of the configuration task defined by the formalization is shown to be **NP**-complete.

However, the language does not allow real number arithmetic. Extending the implementation with these and further aspects of configuration and formalizing a more extensive configuration ontology are important subjects of further work. In addition, the computational complexity of different possible conceptualizations should be further analyzed, and the implementation performance should be tested.

Acknowledgements

Helsinki Graduate School in Computer Science and Engineering has funded the work of the first author, Technology Development Centre Finland the work of the first and third authors, and Academy of Finland (Project 43963) the work of the second author.

REFERENCES

[1] M. Buchheit, R. Klein, and W. Nutt, 'Constructive problem solving: A model construction approach towards configuration', Technical Memo TM-95-01, DFKI, (1995).

[2] A. Felfernig, G. Friedrich, and D. Jannach, 'Uml as domain specific language for the construction of knowledge based configurations systems', in *Proc. of 11th Int. Conf. on Softw. Eng. and Knowl. Eng.*, (1999).

[3] G. Friedrich and M. Stumptner, 'Consistency-based configuration', in *Configuration. AAAI Technical Report WS-99-05*, pp. 35–40, (1999).

[4] T.R. Gruber and R. Olsen, 'The configuration design ontologies and the vt elevator domain theory', *Human-Computer Studies*, **44**, 569–598, (1996).

[5] S. Mittal and B. Falkenhainer, 'Dynamic constraint satisfaction problems', in *Proc. of the 8th Nat. Conf. on AI (AAAI90)*, pp. 25–32, (1990).

[6] I. Niemelä, 'Logic programming with stable model semantics as a constraint programming paradigm', *Annals of Mathematics and Artificial Intelligence*, **25**, 241–273, (1999).

[7] I. Niemelä, P. Simons, and T. Soininen, 'Stable model semantics of weight constraint rules', in *Proc. of the Fifth Intern. Conf. on Logic Programming and Nonmonotonic Reasoning*, pp. 317–331, (1999).

[8] P. Simons, 'Extending the stable model semantics with more expressive rules', in *Proc. of the Fifth Intern. Conf. on Logic Programming and Nonmonotonic Reasoning*, pp. 305–316, (1999).

[9] T. Soininen, E. Gelle, and I. Niemelä, 'A fixpoint definition of dynamic constraint satisfaction', in *Proc. of the 5th International Conf. on Principles and Practice of Constraint Programming*, pp. 419–433, (1999).

[10] T. Soininen and I. Niemelä, 'Developing a declarative rule language for applications in product configuration', in *Proc. of the First Int. Workshop on Practical Aspects of Declarative Languages*, pp. 305–319, (1999).

[11] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a general ontology of configuration', *AI EDAM*, **12**, 357–372, (1998).

Optimizing Configurations

Tommi Syrjänen[1]

Abstract. In this work we examine the problem of finding optimal configurations. The optimality of a configuration is defined with respect to an optimization function. An optimization function imposes a strict partial order on the set of all configurations of a configuration model and optimal configurations are the minimal elements of this relation. We analyze the computational complexity of the problem and define a class of concise optimization functions for which the optimization problem is in $\Delta_2\mathbf{FP}$. We present a formal rule-based language that can be used to express configuration knowledge and show how we can construct concise optimization functions using the language. As a case study, we formalize a subset of the configuration problem for the Debian GNU/Linux system. We define an optimization function for the system and show how it can be used to implement default host profiles.

1 Introduction

In a software configuration management problem, we are given a set of components each providing a distinct functionality and we want to find a collection of these components, a configuration, that provides the functionality that we want. The set of components and their relationships is called a *configuration model*. In a *configuration task* we are given a configuration model and a set of user requirements and we want to find a configuration that satisfies the requirements. In [11] configurations are divided into three classes:

- *valid* configurations satisfy constraints of the configuration model;
- *suitable* configurations are valid configurations that satisfy the user requirements; and
- *optimal* configurations are suitable configurations that additionally satisfy some optimality criteria.

The configuration management problem is widely researched and [8] presents a recent overview of the field. Most of the effort has been expended on finding valid and suitable configurations and optimization issues have received less attention.

The main aim of this work is to define a formal framework for examining the optimization problem. In most cases there is no single optimal configuration as different users have different needs. Thus, we define the optimality of a configuration always with respect to a specific *optimization function*. The basic idea is to establish a preference relation where we prefer a configuration over another if it gives a better value for the function. This relation is a strict partial order over a finite set so it has at least one minimal element. We say that this minimal element is an optimal configuration. It is possible that there is more than one optimal configuration if the function assigns the same value to many configurations.

A configuration system has to be efficient if it is to be used in practice. We define a class of optimization functions, namely *concise* functions, for which the optimization problem is in $\Delta_2\mathbf{FP}$ if the configuration model is described in a formalism that is in **NP**. That is, we can solve the problem with a deterministic Turing machine in a polynomial time if we are given an oracle that solves a **NP**-complete problem in a constant time. The class of concise optimization functions consist of functions that assign each configuration an integral weight less than $a \cdot 2^{n^k}$ where a and k are fixed constants and n is the number of components in the model. This complexity result is proven in Section 7.

Not all configuration formalisms are necessarily **NP**-complete. Soininen and Niemelä [9] examined the complexity of different constraint types in context of their configuration rule language **CRL** and identified the conditions that allow the configuration problem to remain in **P**. It seems that most of the interesting cases are **NP**-complete so we will not give any specific consideration for the problems in **P** in this paper.

We use a rule-based language RRL that is a subset of a language RL defined in [10]. Both languages are based on **CRL**. The problem of finding a suitable configuration for this language is proven to be **NP**-complete in [10].

As a case study, we examine the configuration management of the Debian GNU/Linux system [1]. Debian is a distribution of the GNU/Linux operating system and currently with version 2.1 it has over 2500 distinct software packages that interact with each other in various ways. A package may depend on functionality provided by another one, two packages may conflict with each other, and a package may recommend another package.

In this work we formalize a subset of the Debian configuration problem and show how we can define an optimization weight function for it. In particular, this weight function allows a system administrator to define a set of flexible host profiles that act as default configurations. The complete Debian configuration model is presented in [10].

2 A Configuration Language

In this section we present a declarative rule-based formal language RRL that is a subset of the language RL defined in [10]. The language has a formal semantics that is based on the stable model semantics for normal logic programs [2]. In this work the semantics is presented only informally and the for-

[1] Helsinki University of Technology, Dept. of Computer Science and Eng., Laboratory for Theoretical Computer Science, P.O.Box 5400, FIN-02015 HUT, Finland, Tommi.Syrjanen@hut.fi

mal definition can be found in [10].

The basic language component is an *atom* of the form

$$p(a_1, \ldots, a_n) \ , \tag{1}$$

where p is a predicate symbol and a_1 to a_n are all variables or constants. We will use the convention that all variables start with a capital letter and all constants with a lower case letter. A *literal* is either an atom a or its negation not a. A literal is a *ground literal* if it does not have any variables.

We encode the relationships between atoms using inference *rules* that have two possible forms:

$$h \leftarrow l_1, \ldots, l_n \tag{2}$$

$$\{h_1, \ldots, h_m\} \leftarrow l_1, \ldots, l_n \ , \tag{3}$$

where the atoms h and $h_1, \ldots, h_m$ form the rule *head* and the literals $l_1, \ldots, l_n$ form the rule *body*. The rules of the form (2) are called *basic rules* and the rules of the form (3) are called *choice rules*. A basic rule with an empty rule body is called a *fact*. A RRL *program* is a set of rules.

We are interested in finding stable models of RRL programs. Intuitively, a stable model is a set of atoms that satisfies all rules of the program and each atom in it is justified in the sense that it occurs in a rule head that has its body satisfied in the model. A basic rule asserts that if the body of a rule is true, then the head of the rule has to be true also. On the other hand, if the body of a choice rule is true, any number of head atoms $h_1, \ldots, h_m$ can be in the model. Thus, a choice rule gives a justification for a set of atoms to be in a model but it does not force them to be in it. If an atom does not occur as a head of a rule having a satisfied body, it cannot be true in the model.

In addition to rules and atoms a RRL program may also have *optimize statements* of the form

$$\mathsf{maximize}\ \{p(X) : q(X)\} \ . \tag{4}$$

The notation $p(X) : q(X)$ denotes the set of atoms $p(X)$ for which $q(X)$ is also true. The intuitive meaning of an optimize statement is that we want to find a stable model that has as many of the $p(X)$s true as possible. We can also minimize the number of atoms that are in a model by maximizing their complements:

$$\mathsf{maximize}\ \{\text{not } p(X) : q(X)\} \ . \tag{5}$$

If there are more than one maximize statements in a program, they are considered in fixed order with the first one being the least important and the last one being the most important.

Proposition 1 *The existence of a stable model of a ground* RRL *program is* ***NP****-complete.*

Proof. This proposition is proved in [10]. □

This proposition does not necessarily hold for programs with variables as the semantics of a non-ground RRL program is defined in terms of its Herbrand instantiation which may be exponential in size. However, in most practical cases we can use instantiations that are either linear or quadratic.

3 Configuration Models

A *configuration model* CM consists of a set of objects O_{CM} and a set R_{CM} of relationships between objects. The relationships usually include a set of constraints that define what combinations of objects are legal. The most usual constraint types are:

- an *incompatibility* constraint asserts that two components are mutually exclusive;
- a *dependency* constraint asserts that a component needs another component to work properly; and
- a *choice* constraint asserts that we have to choose one or more components of a given set into the configuration.

In this work a configuration model is defined by a RRL program P_{CM} and there is a direct correspondence between legal configurations of CM and stable models of P_{CM}. We can establish this correspondence by defining a bijection *Conf* that maps sets of atoms to configurations and vice versa. In most cases this function is trivial to define. The reason to separate the model from the program that defines it is that the program may have a set of auxiliary atoms that do not have a direct correspondent in the configuration model. It would also be possible to define a configuration model using other formalisms, for example some of the many constraint satisfaction approaches.

A *configuration* c is simply a subset of O_{CM}. A configuration c is *valid* (denoted $c \models CM$) iff there exists a stable model M of P_{CM} such that $Conf(M) = c$. The set of all valid configurations of CM is denoted $\mathcal{C}_{CM}$.

A set U of *user requirements* is represented by a RRL program P_U. A configuration c is *suitable* (denoted $c \models CM \cup U$) iff there exists a stable model M of a program $P_{CM} \cup P_U$ such that $Conf(M) = c$. The set of all suitable configurations is denoted $\mathcal{S}^U_{CM}$.

4 The Debian Configuration Model

The simplified Debian configuration model presented here contains only the most basic features of the Debian configuration system. We use the available software packages as our configuration objects. There are three relationships in the model:

- *dependency*: A package A *depends on* a package B if A cannot be used at all if B is not installed in the system.
- *conflict*: A package A *conflicts with* a package B if A and B are mutually exclusive.
- *recommendation*: A package A *recommends* a package B if B significantly enhances the functionality of A.

The packages are modeled with facts of the form

$$\mathsf{package(P)} \leftarrow \ . \tag{6}$$

The packages have different priority levels that tell how important a package is to the Debian system. We use here only two priority levels, standard and optional. A package is standard if it should be present in all GNU/Linux systems, otherwise it is optional. We take the approach that we denote a standard package P with a fact

$$\mathsf{standard(P)} \leftarrow \ . \tag{7}$$

All other packages are considered to be optional. The relations between packages are modeled with 2-ary predicates depends, conflicts, and recommends.

The configurations are modeled using the predicate in(P) that is true in a stable model exactly when the package P is chosen to be in a configuration. In effect, we define the *Conf* bijection as:

$$Conf(M) = \{\mathsf{P} \mid \mathsf{in}(\mathsf{P}) \in M\} \tag{8}$$

We will take the approach that a package may be added to a configuration only if we can justify it:

$$\{\ \mathsf{in}(\mathsf{P})\ \} \leftarrow \mathsf{package}(\mathsf{P}), \mathsf{justified}(\mathsf{P})\ . \tag{9}$$

We prune out invalid configurations using two constraints:

$$\leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{not\ in}(\mathsf{P}_2) \tag{10}$$

$$\leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2)\ . \tag{11}$$

The user requirements are modeled using two predicates, user-include(P) and user-exclude(P), for packages that the user specifically wants or does not want to have, respectively. The requirements are added as facts in the program as well as the following two constraints:

$$\leftarrow \mathsf{user\text{-}include}(\mathsf{P}), \mathsf{not\ in}(\mathsf{P}) \tag{12}$$

$$\leftarrow \mathsf{user\text{-}exclude}(\mathsf{P}), \mathsf{in}(\mathsf{P})\ . \tag{13}$$

In the basic model, a package is justified if the user explicitly selected it to be in a configuration, a necessary package depends on it, an included package recommends it, the package is one of the standard packages, or if it belongs to a selected host profile. The host profiles are explained in Section 6.1.

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{user\text{-}include}(\mathsf{P}) \tag{14}$$

$$\mathsf{justified}(\mathsf{P}_2) \leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \tag{15}$$

$$\mathsf{justified}(\mathsf{P}_2) \leftarrow \mathsf{recommends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \tag{16}$$

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{standard}(\mathsf{P}) \tag{17}$$

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{in\text{-}selected\text{-}host\text{-}profile}(\mathsf{P}) \tag{18}$$

Proposition 2 *The existence of a configuration for a Debian configuration model is* ***NP****-complete.*

Proof. By Proposition 1 we only have to prove that the ground instantiation of the configuration model is polynomial with respect to the size of the model.

There are six rules with only one variable in the program and four rules with two variables. The rules with one variable have to be instantiated once for each package in the model and the rules with two variables once for every pair of packages. Thus, if there are n packages in the model, there is a total of

$$6n + 4n(n-1) = 4n^2 + 2n$$

ground instances for the program and its size is polynomial with respect to the size of the model. □

In practice the size of the instantiation is linear since most of the packages are in a relation with less than ten other packages.

5 Optimization Functions

As different users have different requirements, we define the optimality of a configuration always with respect to some optimization function f and there may be many optimization functions defined for a configuration model. We take the approach that optimization functions are defined as weight functions: the function assigns an integral weight for each suitable configuration and we choose the one with highest weight. Formally, we construct a preference relation on the set of all configurations so that we *prefer* a configuration c_1 over c_2 if c_1 has a higher weight than c_2.

Definition 1 *Given a configuration model CM and a weight function $f : \mathcal{C}_{CM} \to \mathbb{N}$, the* preference relation $P(CM, f)$ *is the relation*

$$P(CM, f) = \{\langle c_1, c_2\rangle \mid c_1, c_2 \in \mathcal{C}_{CM}, f(c_1) > f(c_2)\}\ . \tag{19}$$

The Definition 1 imposes a strict partial order on all valid configurations of the model. Given a set of user requirements, we can restrict this relation to suitable configurations.

Definition 2 *Given a configuration model CM, a set U of user requirements, and a weight function $f : \mathcal{S}^U_{CM} \to \mathbb{N}$, the* restricted preference relation $R(CM, U, f)$ *is the relation*

$$R(CM, U, f) = \{\langle c_1, c_2\rangle \mid c_1, c_2 \in \mathcal{S}^U_{CM}, f(c_1) > f(c_2)\}\ . \tag{20}$$

An optimal configuration is a minimal element of the restricted preference relation.

Definition 3 *Given a configuration model CM, a set U of user requirements, and a function $f : \mathcal{S}^U_{CM} \to \mathbb{N}$, a configuration $c \in \mathcal{S}^U_{CM}$ is an* optimal configuration with respect to f *if and only if c is a minimal element of $R(CM, U, f)$.*

Alternatively, we could try to leave out the definition of the restricted preference relation and demand only that a configuration has to be both suitable and a minimal element of $P(CM, f)$. However, this would cause a problem when P has no suitable minimums.

The definitions above do not impose any requirements on the weight function f. In practical applications the weight function should be computationally feasible. We define a class of functions, namely concise weight functions, that has the property that the optimization problem for them is in $\Delta_2\mathbf{FP}$. This will be proved in Section 7.

Definition 4 *A function $f : \mathcal{C}_{CM} \to \mathbb{N}$ is a* concise *weight function if and only if there exist two constants, $a, k \geq 0$, such that for all configurations c, $f(c) \leq a \cdot 2^{n^k}$, where n is the number of components in the configuration model, and f is computable in a polynomial time.*

Both requirements are necessary. The upper bound of the value ensures that we have to consider only polynomial number of potentially optimal weights the time bound ensures that we can compute the value efficiently.

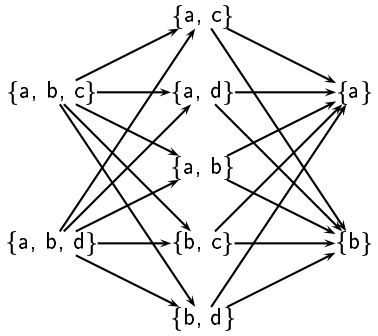


Figure 1. The preference relation of Example 1

Example 1 Let a configuration model CM be defined by the following program:

$$\begin{aligned} \{\ \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\ \} &\leftarrow \\ &\leftarrow \mathsf{c}, \mathsf{d} \\ &\leftarrow \mathsf{not\ a}, \mathsf{not\ b} \end{aligned} \tag{21}$$

In addition, let a function $f : \mathcal{C}_{CM} \rightarrow \mathbb{N}$ be defined as follows:

$$f(c) = |c| \ . \tag{22}$$

Now CM has nine valid configurations:

$$\begin{aligned} \mathcal{C}_{CM} = \{\ &\{\ \mathsf{a}\ \}, \{\ \mathsf{b}\ \}, \{\ \mathsf{a}, \mathsf{c}\ \}, \{\ \mathsf{a}, \mathsf{d}\ \}, \{\ \mathsf{b}, \mathsf{c}\ \}, \\ &\{\ \mathsf{b}, \mathsf{d}\ \}, \{\ \mathsf{a}, \mathsf{b}\}, \{\ \mathsf{a}, \mathsf{b}, \mathsf{c}\ \}, \{\ \mathsf{a}, \mathsf{b}, \mathsf{d}\ \}\ \} \end{aligned} \tag{23}$$

The preference relation $P(CM, f)$ is presented in Figure 1. There are two configurations in $\mathcal{C}$ that are optimal with respect to f, namely $\{\ \mathsf{a}, \mathsf{b}, \mathsf{c}\ \}$ and $\{\ \mathsf{a}, \mathsf{b}, \mathsf{d}\ \}$. Suppose that the user wants to have the component d in the configuration so we only allow configurations where d is present:

$$\mathcal{S}^U_{CM} = \{\ \{\ \mathsf{a}, \mathsf{d}\ \}, \{\ \mathsf{b}, \mathsf{d}\ \}, \{\ \mathsf{a}, \mathsf{b}, \mathsf{d}\ \}\ \} \ . \tag{24}$$

The unique optimal configuration is now $\{\ \mathsf{a}, \mathsf{b}, \mathsf{d}\ \}$. ◊

6 Defining Weight Functions with RRL

We can define concise optimization functions using maximize statements of RRL. The functions are defined in terms of configuration objects by means of *optimization criteria.*

Definition 5 *An* optimization criterion C *is a set*

$$C = \{o_1, \ldots, o_n\} \tag{25}$$

of configuration objects.

We will define the weight function is such way that the more objects $o_1, \ldots, o_n$ are true in a stable model of the RRL program, the higher weight it assigns to the corresponding configuration. If there are more than one criteria, they are ordered in an ascending order of importance and the more important a criterion is, the higher weight it assigns to a configuration. We also allow the criteria to be defined dynamically, during a configuration task.

Definition 6 *Given a configuration c and an ordered tuple $C = (C_1, \ldots, C_k)$ of optimization criteria, the* weight function *$w_k(c)$ is defined as follows:*

$$w_k(c) = \sum_{i=1}^{k} n^{2i} \cdot |C_i \cap c| \tag{26}$$

Theorem 1 *The weight function $w_k(c)$ is concise.*

Proof. Let n be the number of components in the configuration model and k the number of optimization criteria. The weight function $w_k(c)$ is clearly computable in polynomial time. The maximum value of $|C_i \cap c|$ is n so the value of$w_k(c)$ is always strictly less than n^{2k+2} when $n > 1$. Thus, we can assign an upper bound $2^{n^{2k+2}}$. If $n = 1$, then $w_k(c) \leq k$ and we can use $2k$ as a bound so $w_k(c)$ is concise. □

The optimality criteria $C_1, \ldots, C_k$ can be defined with k maximize statements:

$$\begin{aligned} &\mathsf{maximize}\ \{\mathsf{in}(X) : C_1(X)\} \\ &\qquad\vdots \\ &\mathsf{maximize}\ \{\mathsf{in}(X) : C_k(X)\} \end{aligned} \tag{27}$$

The main problem of this formulation is that we may use it only when the optimality can be expressed in terms of objects that are in the configuration. In other cases we would like to be able to use a more general weight function. This can be achieved by allowing any literals to be in a criterion and extending the maximize statements to have the form:

$$\mathsf{maximize}\ \{p(X) : q(X) = g(p(X))\} \tag{28}$$

where $g(p(X))$ is an integer-valued function that assigns a weight to each literal $p(X)$. The idea is then to maximize the total weight of the literals. However, the function g has to be defined carefully to ensure that the complete weight function stays concise.

6.1 The Debian Weight Function

The configuration problem of a GNU/Linux system has two important characteristics:

1. the configuration model is large and there are usually hundreds of packages in a configuration; and
2. the GNU/Linux is free software so a configuration with a hundred packages has essentially the same cost than one with a thousand packages.

Because of the first property, configuring the system may be very tedious and it may take many hours if the user has to select packages one at a time. This could be simplified by constructing a tool that allows the user to state in general terms what he or she wants to do with the system. The tool would then automatically generate a configuration satisfying the user's needs. We can do this by defining collections of related packages, *host profiles*, that can be selected in addition to normal packages. The profiles can be defined either by the creators of the distribution or by a local system administrator.

Because of the second property of the problem domain, we would often want to include potentially useful packages into the configuration if there is enough disk space available for them. Installing them at the same time when the other system is configured saves time and trouble in the long run.

We define the optimization criteria for the simplified Debian configuration model using the following principles:

1. maximize the number of packages belonging to a host profile;

2. maximize the number of packages that have standard priority; and
3. maximize the number of packages that are recommend by other packages that are in the configuration.

We can see that these criteria sometimes impose conflicting demands. For example, it is possible that the host profile includes a package that is in conflict with a standard package and satisfying the first criterion makes it impossible to satisfy the second criterion.

We order the criteria so that we prefer adding one package from the host profile over any number of standard packages or recommended packages and one standard package over any number of recommended packages. This is by no means the only possible ordering of the criteria.

Conceptually, during a configuration task we first add packages that are needed to satisfy the user requirements. Next, we add packages that are in the host profile and do not conflict with packages that are already in the configuration. After that, we add all standard packages that do not cause problems. Finally, we try to find a maximal set of packages that are recommended by those already in the configuration and add them to it.

The Debian weight function can be defined with the following maximize statements:

$$\begin{aligned} &\mathsf{maximixe}\ \{\ \mathsf{in(P)}\ :\ \mathsf{is\text{-}recommended(P)}\ \} \\ &\mathsf{maximize}\ \{\ \mathsf{in(P)}\ :\ \mathsf{standard(P)}\ \} \\ &\mathsf{maximize}\ \{\ \mathsf{in(P)}\ :\ \mathsf{in\text{-}selected\text{-}host\text{-}profile(P)}\ \} \end{aligned} \tag{29}$$

The maximize statements are again in an ascending order of importance. The predicate in-selected-host-profile is true when the package in question belongs to an included host profile and the predicate is-recommended is defined as follows:

$$\mathsf{is\text{-}recommended(P_2)} \leftarrow \mathsf{recommends(P_1, P_2), in(P_1)}\ . \tag{30}$$

7 Computational Complexity

In this section we define a set of configuration tasks and analyze their computational complexities. The complexity results suppose that the configuration models are defined with RRL but the results generalize to other **NP**-complete formalisms.

We show that if we have a variable-free configuration model, the validity of a configuration can be checked in a polynomial time with respect to the size of the model. The problem of finding a suitable configuration given a set of user requirements is **FNP**-complete as is the problem of finding a suitable configuration with a specific weight if the weight function f is concise. We show that the problem of finding an optimal configuration is in $\Delta_2\mathbf{FP}$. It is not clear whether the problem is also $\Delta_2\mathbf{FP}$-complete.

The results are stated for variable-free configuration models but they also hold it the size of the instantiation of the model is polynomial, which is true for most practical cases.

Definition 7 *Given a ground configuration model CM and a configuration c, the problem of checking whether $c \models CM$ is called* VALID-CONFIGURATION(CM, c).

Theorem 2 *The problem* VALID-CONFIGURATION(CM, c) *is in* ***P****.*

Proof. The theorem follows directly from Proposition 1. □

Definition 8 *Given a ground configuration model CM and a set U of user requirements, the problem of finding a configuration c such that $c \models CM \cup U$ is called* QUERY-CONFIGURATION(CM, U)

Theorem 3 *The problem* QUERY-CONFIGURATION(CM, U) *is* ***FNP****-complete.*

Proof. The theorem follows from Proposition 1. □

Definition 9 *In the problem* QUERY-CONFIGURATION-WITH-WEIGHT(CM, U, f, W) *we are given a ground configuration model CM, a set U of user requirements, a weight function f, and a weight $W \in \mathbb{N}$ and the object is to find a configuration c such that $c \models CM \cup U$ and $f(c) \geq W$.*

Theorem 4 *If the weight function f is concise, the problem* QUERY-CONFIGURATION-WITH-WEIGHT(CM, U, f, W) *is* ***FNP****-complete.*

Proof. Follows directly from Theorem 3 and the fact that we can guess the weight of a configuration and verify it in a polynomial time by Definition 4. □

Definition 10 *Given a ground configuration model CM, a set U of user requirements, and a weight function f, the problem of finding a configuration c such that $c \models CM \cup U$ and c is optimal with respect to f is called* OPTIMAL-CONFIGURATION(CM, U, f).

Theorem 5 *If the weight function f is concise, the problem* OPTIMAL-CONFIGURATION(CM, U, f) *is in $\Delta_2\mathbf{FP}$.*

Proof. By definition [6, p.424], a problem is in $\Delta_2\mathbf{FP}$ when it can be solved in deterministic polynomial time using a **NP**-oracle. As the weight function f is concise, the optimum weight is at most 2^{n^k} for some fixed n and k. We can find the weight of an optimal configuration by using *binary search*. We first check whether there exist a suitable configuration that has a weight 2^{n^k-1} or more. As this is an instance of QUERY-CONFIGURATION-WITH-WEIGHT, it can be done in constant time using the oracle.

Now, depending on the answer we either check whether the optimum weight is over 2^{n^k-2} or over $2^{n^k-1} + 2^{n^k-2}$. Thus, in each step we halve the possible range of the optimal weight and we have to issue $\log_2 2^{n^k} = n^k$ queries so the problem is in $\Delta_2\mathbf{FP}$. □

8 Implementation and Evaluation

The Debian configuration model has been implemented as an extended logic program using the SMODELS system [5] developed in the Laboratory for Theoretical Computer Science in Helsinki University of Technology. The SMODELS system is available at

`http://www.tcs.hut.fi/pub/smodels` .

The full Debian configuration model with preferences was tested using actual data of the main Debian distribution 2.1 with 2255 different packages. The user requirements were generated by choosing randomly a fixed number of packages. The number of user requirements varied between 0–100. In addition, a host profile with 89 packages was created and included in the configuration. The tests were run under Debian GNU/Linux 2.1 on a 450 MHz Intel Pentium III system with 256 MB main memory. The used software versions were `smodels-2.25` and `lparse-0.99.48`.

The results of the tests are shown in Table 1. The times shown are averages over 100 runs. Separate figures are presented for satisfiable and unsatisfiable user requirements. The times show only how long `smodels` took to find an optimal configuration. In addition to this time, `lparse` spent on average 9.1 seconds preprocessing the model. This time is not shown in the figure since in a practical configuration tool this preprocessing has to be done only once for each Debian version. The execution times get faster as the number of user requirements increase because adding new constraints reduces the number of suitable configurations.

Table 1. Evaluation results

#Pck.	Total (s)	Sat. (s)	Unsat. (s)	#Sat
0	2.94	2.94	–	100
5	2.66	2.78	1.43	91
10	2.38	2.69	1.02	82
50	1.80	2.26	1.02	62
75	1.56	2.17	1.02	47
100	1.31	2.11	1.03	26
total	2.11	2.61	1.04	68

9 Related Work

Most of the existing work on optimal configurations has been concentrated in the area of *resource balancing*. In a resource balancing problem we are given a set of *consumers* that each consume a specific resource. There are also several *producers* that generate the resources. The goal is to find a set of producers that generates the resources that consumers use. Each producer has also a distinct *cost* assigned to it and we want to minimize the total cost of producers that are taken into the configuration.

Sabin and Freuder [7] modeled the problem using composite constraint satisfaction. Their basic idea is to achieve optimality by adding a new constraint to the system each time a new solution is found. If the found solution has a cost C, the constraint $cost < C$ is added to the program. This forces the underlying engine to find only better solutions. The approach is complete and it will always find the optimal configuration.

Juengst and Heinrich [3] took another route by proposing a heuristics that would find good, but not necessarily optimal, configurations. They compute the ratio of actual resource demand and highest possible resource supply in a local scope, and try to keep it as low as possible during the configuration process.

Conceptually the approach of Sabin and Freuder fits within our definition of an optimal configuration. The RRL is not well suited for resource balancing programs but if it is extended with weight constraint rules [4], it gains sufficient expressive power to allow a relatively compact representation of the problem.

10 Conclusions and Future Work

We presented a formal definition for optimal configurations. The basic approach is to define an optimization function that imposes a preference relation on the set of all configurations. The optimal configurations are minimal elements of this relation. We defined the class of concise weight functions for which the optimization problem is in $\Delta_2\mathbf{FP}$.

We presented a rule-based language RRL that can be used to define configuration models and showed how optimization functions can be defined with it.

We formalized a subset of the configuration management of the Debian GNU/Linux system and defined a concise weight function for it. The weight function allows a system administrator to define a set of easily customizable host profiles. The basic idea is that first the user requirements are satisfied and then as many default packages are added to the configuration as possible. The full configuration model for the Debian system is presented in [10]. The model has not been incorporated into any existing tool and the next step of this research is to construct a working prototype.

ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland (project no. 43963) and Helsinki Graduate School in Computer Science and Engineering (HeCSE). We would also like to thank the referees for helpful comments.

REFERENCES

[1] Debian GNU/Linux. Available at: <url:http://www.debian.org>.

[2] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[3] Werner E. Juengst and Michael Heinrich. Using resource balancing to configure modular systems. *IEEE Intelligent Systems & their applications*, pages 50 – 58, October 1998.

[4] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the Fifth Interational Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, December 1999.

[5] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, April 2000.

[6] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc, 1995.

[7] Daniel Sabin and Eugene C. Freuder. Optimization methods for constraint resource problems. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[8] Daniel Sabin and Rainer Weigel. Product configuration frameworks — a survey. *IEEE Intelligent Systems & their applications*, pages 42 – 49, October 1998.

[9] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*. Springer-Verlag, January 1999.

[10] T. Syrjänen. A rule-based formal model of software configuration. Research Report A 55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 1999.

[11] B. Wielinga and G. Schreiber. Configuration-design problem solving. *IEEE Expert*, 12 2:49 – 56, March-April 1997.

IPAS: An Integrated Application Toolsuite for the Configuration of Diesel Engines

Dr. Carlo Bach [1]

Abstract. In the presentation the application toolsuite IPAS for configuring and offering diesel engines will be demonstrated.

While IPAS provides a complete set of applications for the quotation and order taking process we will mainly focus on the configuration process as seen by the end user and the product model developer. The highlights of the configurator as parallel configurations and reconstruction of old configurations will be explained. An integrated development environment to build up, test, and maintain product models is described as well.

Experiences made during the project will be discussed.

1 THE IPAS PROJECT

Motoren- und Turbinen Union GmbH Friedrichshafen (MTU [3]) in Germany, together with its subsidiaries, forms the Diesel Propulsion Systems Division in the DaimlerChrysler Group and is one of the worldwide leading manufacturers of large diesel engines and complete drive systems. Diesel engines in the 35 to 7,400 kW power range are manufactured for marine propulsion, heavy vehicles, power generation and railroad applications.

In order to handle the growing demand for customer specific configured diesel engines MTU initiated a project called IPAS (Integriertes Projekt Abwicklungs System) in 1996. IPAS provides a complete suite of applications to handle the quotation and order taking process. At its heart an interactive configurator masters the technical complexity of the diesel engines.

In late 1998 the applications went into productive use at the headquarter. Meanwhile several improvements especially in the management of the configuration knowledge could be realized. Currently the system is going to be deployed at MTU's subsidiaries world-wide. The tool suite is in daily use by about 70 users.

1.1 The Application Domain

MTU offers not only single diesel engines but complete propulsion systems for various application domains. As illustrative example we will concentrate in the following, however, on a marine application.

A typical quotation for a shipping company handles a fleet of two to four ships each consisting of several diesel engines and controlling equipment. E.g. a ship needs normally two engines for propulsion, one engine for power generation and computer systems for steering. All these engines have to be configured differently to cope with the individual requirements.

[1] University of Applied Sciences Buchs, Departement of Computer Science, Werdenbergstrasse 4, CH 9471 Buchs, Switzerland, email: carlo.bach@ntb.ch, www.ntb.ch

For example the location of an engine in the hull - wether it is on the left or the right side - decides on which side the oil dipstick has to be mounted (right and left). Of course, an oil dipstick on the left side of the engine is a different part than one on the right side and the engines itself - although very similar - are quite different in terms of the parts involved.

The configuration results in a hierarchical parts list which can be calculated and printed as Microsoft Word documents.

As this small example shows, the engines have a great variance and are therefore quite complex. A single engine consists in a final configuration of about 400 parts out of several thousand possible parts.

Besides of the technical complexity an offering process may take from some weeks to several months or even years. During this period of time already configured engines have to be reconfigured because of new customer requirements or since the diesel engine type itself has changed technically. Therefore adaptation of old configurations is an integral element of the tool suite.

To handle this complexity MTU decided to develop a quotation system consisting of a configurator and all the applications needed to smoothly deal with the quotation and ordering process.

1.2 Application Overview

IPAS provides sales people at MTU with the following six highly integrated applications:

- PO: is the so called project folder (Projekt-Ordner). It offers a hierarchical view on projects, subprojects and all the documents generated by the other applications. PO allows to access project information at all subsidiaries and is responsible for the distribution of the relevant application data between different locations. It also stores versions of quotations and orders.
- ADMIN: is used for the management of customer and project information common to all applications.
- CONFIG: is the main application for the technical configuration of ships with several engines. It is used alike for quotes and for orders.
- CALC: calculates prices based on the detailed technical configuration generated with CONFIG and prints detailed price lists.
- TERMS: prepares a written offer based on text components. Various parameters can be extracted from the technical (CONFIG) and the commercial (CALC) configuration.
- NOTE: manages the order taking, the production, and the delivery of the ordered engines at MTU itself.

To work properly these end user applications need a lot of basic data which are maintained with several other applications.

The central application for the management, test, and maintenance of the configuration knowledge is PMADMIN (Produkt-Modell-Administrator).

1.3 Application CONFIG

With CONFIG sales people can interactively configure diesel engines for a fleet of ships and manipulate them afterwards. The application supports the following main features:

- The user has first to select an engine type which will determine the prodcuct model. Each product model exists in several versions depending on how much it changed over time.
- CONFIG presents always a complete and technical correct solution as far as the configuration knowledge is valid. Choices technically forbidden by the configuration knowledge can not be selected. All selections are performed really interactively in typically less than one second.
- The user can configure the engines based on a sales model and/or on an engineering model. The sales model lists about 30 to 60 easy to understand properties each of them with several alternatives. In the engineering model 2000 to 6000 parts describe the detailed assembly of an engine with all its variations.
  It is always possible to switch between these two views since the configuration engine propagates changes in both directions.
- The configuration engine supports the parallel configuration of diesel engines. By default all engines are configured together. But the user can deliberately select a subset of engines and can even change the set any time during the configuration process.
  An individual instance of the configuration engine is created for each element in the set of selected engines. The user interacts with a so called leading engine in the set. All changes in the leading engine are duplicated whenever possible in the other configurations in the set. Conflicts are reported to the user.
  This feature reduces the configuration time for large quotations dramatically.
- The user can modify existing or insert new components in the product model on a per offer basis. The configuration engine integrates them for the current user session into the product model although it can not test if they are appropriate and technical compatible with the product model.
- Configurations can be reconfigured at any time, e.g. if a customer calls for modifications.
- The configuration engine can adapt configurations based on older product models to new models. Conflicts are reported to the user.
  The configuration engine uses the following method for this reconstruction process: When a configuration is stored an identifier of the product model used and only the explicit choices made by the user are written to the data store. All selections made by the system are left aside.
  If a configuration has to be restored with the same product model as stored the configuration engine selects first the choices done by the user last time. Afterwards all other selections are done automatically again. In the case just described the reconstructed configuration is absolutely the same as the original one and no conflicts can arise since the same unmodified product model is used.
  The same mechanism is used for the reconstruction with a different version of the prodcut model (typically a more actual one). In this situation, however, several conflicts can occur.
  First, if a choice is not available in the new product model any more this conflict situation is written to a log and the system will postpone the decision upon which variant should be selected until all other choices are reconstructed.
  A second conflict situation has to handle choices of parts which are not available at the same place in the parts list hierarchy but have moved. In this situation the system refines the configuration stepwise and tries to find the choice after each step again.
  Finally, the original configuration is compared to the reconstructed one. All differences are written to the log and presented to the user.
  This reconstruction mechanism works best if product models change only slowly over time. Since it is based on the naming of properties and components it is important to keep the naming conventions while the product models evolve.
  This feature is mainly used for long running quotations or orders where ships are delivered over a long period of time, e.g. one ship per year.

1.4 Application PMADMIN

PMADMIN is an integrated development environement for the management, test, and maintenance of product models. It is used mainly by the diesel engine engineers who have deep knowledge about their products.

Since the creation of product models is similar to the programming of application programs PMADMIN provides a lot of features analogous to those known to software engineers from visual development environements. The users of PMADMIN, however, are not very familiar with software construction. Therefore the application provides all tasks in a visual attractive fashion.

The product model used by CONFIG is a result of a transformation into the configuration engines own language and a compilation process.

PMADMIN supports the following features:

- Creation and modification of product models: PMADMIN provides three categories of components: properties, parts, and rules. Each category represents one aspect of configuration knowledge for the diesel engine engineers, although internally these categories are not differentiated.
- Modularization: Product models are divided into so called libraries. Libraries can be reused.
- Versioning: PMADMIN keeps track if product models are used in active configurations. These product models can not be changed any more since the reconstruction without conflicts could not be guaranteed.
  If changes are necessary PMADMIN creates a new version of the product model keeping as much parts unchanged as possible. A new version has to be explicitly released for use.
  To get a fast overview PMADMIN can show differences between product models.
- Views: Several views on the configuration knowledge are possible. E.g. a user can look at different aspects of the product model like the sales and the engineering model.
- Searching: PMADMIN provides extended browsing, searching, and crossreference capabilities. Since the product models grow very fast (up to 6000 components) good filtering and searching techniques are needed to keep an overview of the parts involved.
- Run-time debugging: With the run-time debugger the developper can follow in detail the configuration process. Whereas in CONFIG rules are not shown to the user in PMADMIN this configuration knowledge is available.
  It is always possible to look at the current solution state and the dependency information.

Since the product models are very large stepwise debugging is normally to time consuming. Therefore, the debugger supports sophisticated break points and watches statements.

2 CONFIGURATION ENGINE

IPAS uses a domain independent configuration engine [1], [2]. The knowledge representation language applies techniques known from constraint, logic, and object-oriented programming. The language is completely declarative. As basic language constructs aggregation, generalization, and constraints similar to [4] are applied.

The inference engine searches an AND/OR-lattice with a backtracking method. To improve the search efficiency problem dependent heuristics can be employed additionally.

In case of conflicts a dependency based backtracking algorithm is called, minimizing the changes in the current solution and keeping as much user choices as possible.

3 EXPERIENCES

The applications in IPAS evolved over the project period and the configuration process was refined iteratively several times as end users but also product model developers gained experiences with configuration tasks.

It has shown that powerful and flexible configuration engines are needed to react in a timely manner to changed user requirements. In our case it was used to adapt the configuration process as seen by the end user without the need to change the configuration engine itself.

Another lesson learned is that the power of a configuration engine can be overwhelming to the product model developers as well as the end users. It is therefore absolutely necessary to provide product model developers with programming conventions to guide the development process.

We introduced naming conventions which not only improved the communication between developers but also had a positive effect on the reconstruction process in CONFIG. We also provided predefined patterns for typical configuration problems. Most important, these rules are supported by PMADMIN either in form of visual entry masks or as check routines.

Nevertheless the full power of the configuration engine can always be exploited if the need arises.

4 IMPLEMENTATION

The whole IPAS system is realized as a multi-user distributed client-server system. It runs under Microsoft Windows 95, 98 and NT. The front-end applications are written in Visual Basic and access data stored on a Microsoft SQL-Server 7.0 relational database. For printing an integration to Microsoft Word was implemented. The configuration engine is written in C++ and is available on Windows and Unix plattforms.

5 CONCLUSION

The IPAS system consists of a complete application toolsuite for the fast and reliable management of quotations and orders in a technical complicated domain. At its heart stands a powerful domain independent configuration engine.

As in the software development process programming conventions are necessary to master the complexity of the configuration task.

The IPAS system is in daily use at MTU for almost two years now.

The configuration engine was not designed exclusively for the IPAS system but is used in other applications as well, e.g. it also configures switching equipement in a batch process at the mobile networks division of a big Telecom company.

ACKNOWLEDGEMENTS

I would like to thank Michael Feldmann and the computer science departement from MTU Motoren- und Turbinen-Union, Friedrichshafen.

REFERENCES

[1] ISE Software AG. Configuration Master User's Manual.
[2] C. Bach, *An Interactive Knowledge-Based Shell for Configuration Tasks*, Hartung-Gorre Verlag, D-Konstanz, 1994. also published as Ph.D. thesis, ETH Zuerich.
[3] MTU Friedrichshafen. www.mtu-friedrichshafen.com.
[4] H. Meyer auf'm Hofe, 'Construct: Combining concept languages with a model of configuration processes', in *Papers from the AAAI Workshop on Configuration*, (1999).

EngCon
A Flexible Domain-Independent Configuration Engine

Oliver Hollmann, Thomas Wagner[1] and **Andreas Guenter**[2]

Abstract. The knowledge-based configuration engine *EngCon* is presented which is developed in a cooperation of TZI Bremen and LENZE GmbH & Co KG Hameln, Germany.
We discuss the system architecture and the configuration techniques used in *EngCon.* In our demonstration we try to focus on the differences to the LISP prototype *KONWERK* which was developed at the University of Hamburg, Germany.

1 INTRODUCTION

The configuration of products with a wealth of variants has enormously gained in relevance over the last years. Customers increasingly expect consideration of their individual requirements on a product. There was a paradigm switch from *mass production* to *mass customization.* Intelligent configuration and design tools support sales managers, engineers, and customers to layout and configure individual, innovative, and complex products.
The implementation of a knowledge-based configuration engine in a company leads to more transparency and efficiency in the usage of expert knowledge. The specific configuration knowledge of a domain expert can be used by colleagues or customers directly.
In the automobile industry, for example, a customer can configure a car directly on the internet page of a company[3] [4] and can order it without any personal contact to a dealer. An overview of online configurators is given in [Boehm et al., 1999].
In this paper we discuss the domain-independent, knowledge-based configurator *EngCon.* The main idea of the tool is methodically based on the LISP prototype *KONWERK* (see [Guenter, 1995]). But in detail there are some differences and extensions.
The next section provides a short introduction to knowledge-based configuration. Section 3 shows the configuration engine *EngCon.* We discuss the knowledge representation, problem-solving methods, architecture and system integration with CRM- and ERP-systems. Finally, a short summary and some future directions of *EngCon* are given in section 4.

2 KNOWLEDGE-BASED CONFIGURATION

Solving a configuration task means selecting, parametering, and composing a system from single components to a valid solution according to all requirements and constraints [Sabin, 1998] [Guenter, 1995]. A configuration task consists of a problem representation and algorithms generating a solution. There are several problem-solving methods for configuration problems (e.g. logic-based, structure-oriented, ressource-oriented, associative and function-oriented) which were discussed in [Brinkop, 1999] and [Guenter, 1995].
The domain-independent configuration tool *EngCon* [Arlt et al., 1999] works with the structure-oriented configuration method. The tool is successfully used for the complex product configuration of electronic drives. *EngCon* is implemented in Java and has an ontology for representing domain knowledge. Domain concepts can be defined with parameters and relations in a concept hierarchy. Modeling descriptions for taxonomic (is-a), partonomic (has-part) and user-defined relations can be used. The action of the configuration engine can be controlled with declarative control knowledge. Requirements and restrictions between configuration objects are represented with constraints.

3 ENGCON TECHNOLOGIES

The domain-independent, knowledge-based configurator *EngCon* is successfully used for the configuration of electronic drive systems. The structure-oriented configuration method is well suitable for technical domains with a wealth of variants and relations between the components. The user can develop step by step the solution guided by the configurator (see figure 1). In this section we discuss the technologies of *EngCon* in detail.

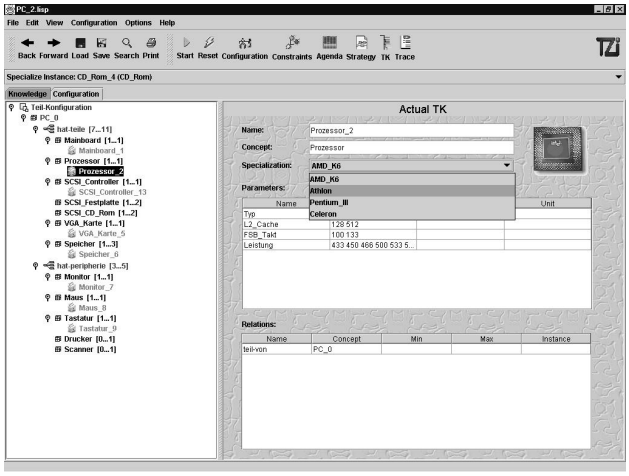


Figure 1. EngCon System

[1] TZI, Center for Computing Technologies, University of Bremen, D-28359 Bremen, Germany, email: {oho,twagner}@tzi.de
[2] HITeC e.V., University of Hamburg, D-22527 Hamburg, Germany, email: guenter@informatik.uni-hamburg.de
[3] http://www.opel.de/webkauf
[4] http://www.bmw.de/carconfigurator

3.1 Knowledge Representation

The domain knowledge is represented in an object-oriented concept-hierarchy. Domain objects are defined with parameters and relations. *EngCon* provides the *taxonomic* (is-a) and *partonomic* relation (has-part).

```
(def-do
  :name <name>
  :super-concept <name>
  [:parameters (<parameter>[<facette>]*)]
  [:relations  (<relation>[<facette>]*)])
```

A facette might be for example a default value or a measurement unit. The representation of concepts is equal to the frame-based representation in *KONWERK* (see [Guenter, 1995]).
The configuration process in *EngCon* can be structured in different phases which are described by control knowledge (see [Guenter, 1992]). In a strategy we can focus the agenda generation on specific configuration objects. The processing of the agenda can be controlled with agenda selection criteria. You can define a stack of calculation methods (see subsection 3.2) for the evaluation of a configuration step. Strategies are changed if the current agenda is empty or the start condition of a strategy with a higher priority is true.
The control knowledge of *EngCon* describes the funcionality of an intelligent assistant. A user can configure either interactively or guided by the tool.
Other than in *KONWERK*, the look and feel of configuration objects in a user interface can be defined with declarative presentation concepts:

```
(def-presentation
  :name <name>
  :super-presentation <name>
  :concept <concept-name> | <task-name>
  [:strategy <strategy-name>]
  [:parameters (<parameter>*)]
  [:relations  (<relation>*)])
```

A parameter of a presentation concept might be for example an icon of a component which should be presented for a configuration object in a specific situation of the configuration process. Depending on an interacting user, it is possible to describe different configuration modes (e.g. novice and expert) and present detailed informations.

3.2 Problem-Solving Methods

The configurator *EngCon* works agenda-based and provides several calculation methods to determine the value in a configuration step. In a top-down design a configuration object can be *parametrized*, *specialized* and *decomposed* into parts. In a bottom-up or mixed design, components can be *integrated* in complex configuration objects, too. A configuration decision can be made by:

- user-interaction
- default-value-takeover
- dynamic-default-calculation (e.g. min/max value)
- calculation functions
- constraints
- taxonomic inferences

Constraints are used to check the consistency of a partial solution and to restrict and propagate values. There are different constraint relations provided in *EngCon* (tuple, function, java, specialize, decompose) which are defined in a conceptual way. The bindings of constraint pins are defined with variable-pattern-pairs. The constraint relations are instantiated incrementally during the configuration process. Only the constraints for actual configuration objects are in the scope of the constraint propagation.

```
(def-conceptual-constraint
  :name <name>
  :variable-pattern-pairs ((<var><pattern>)*)
  :constraint-calls ((<cons><var><slot>)*))
```

Tuple constraints can be defined directly in a relational database system. The evaluation is done by SQL-queries via JDBC.
It is possible to *specialize* or *decompose* configuration objects automatically via constraints.
JAVA constraints are a flexible and powerful way to define customized constraint relations. The functionality of a constraint can be implemented in a JAVA method with an clear API to the *EngCon* constraint-solver. With this powerful mechanism the availability of a component for example can be checked through a request to an ERP-system.
If the values of a configuration object are restricted in a way that there is only one possible specialization in the concept hierarchy, the specialization step will be automatically done by the taxonomic-inference mechanism of *EngCon*. Complex configuration objects will be decomposed automatically to the minimum number of components defined in the knowledge base.

3.3 Integration

The concept hierarchy of *EngCon* corresponds to a company's product model. In the knowledge acquisition process the concept definitions for the bottom levels of the concept hierarchy can be generated automatically from the database of an ERP-system (see figure 2). The higher levels of the concept hierarchy have to be modeled by a domain expert and knowledge engineer.

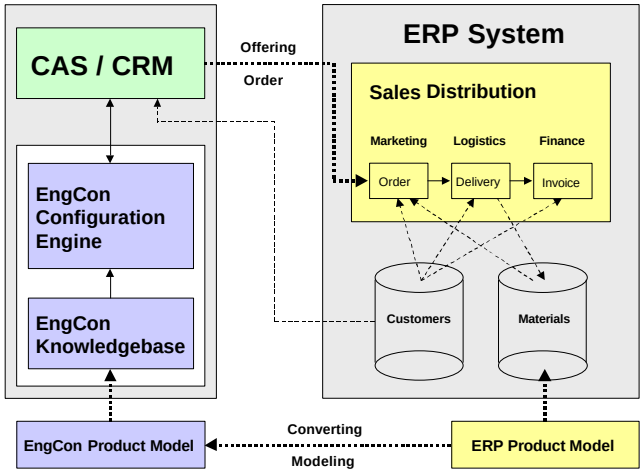


Figure 2. EngCon ERP-Integration

The solution of the configurator can be directly integrated in a supply or an order. *EngCon* provides interfaces to CRM-systems. The configuration process can be initialized by a CRM-system and the

output can be exported and included in the supply text or order text.
A closer integration with an ERP-system of a company can be achieved by the implementation of JAVA constraints which initiate specific transactions (see subsection 3.2).

3.4 Implementation

The configurator *EngCon* has a flexible component architecture based on JAVA-2 technology. The platform-independent JAVA technology allows the usage of the configurator in heterogeneous software environments and provides a good basis for internet configuration. The interface of *EngCon* can be bridged for example with JAVA-RMI, and the user interface might be an Applet.
The kernel of *EngCon* can be integrated in domain-specific user interfaces. It is possible to embed the configurator via middleware technology in non-JAVA environments.
In an early phase of our project we integrated the configurator via SUN Microsystems *ActiveXBridge* in a Visual Basic user interface.

4 SUMMARY AND FUTURE DIRECTIONS

The configurator *EngCon* is successfully used in technical domains. A future extension might support 3-D configurations with spatial constraints.
Another task for the future will be the storage of solutions and their *case-based retrieval.* Time-intensive configuration processes can be saved, if the tool provides traditional solutions to a configuration task which can be reconfigured.
We are actually working on an online solution for the internet. Therefore the knowledge representation should be transformed to XML which seems to be the standard representation in eCommerce. There are several XML-approaches to represent product data in electronic catalogues[5] or to define interchange formats[6] which can be integrated in *EngCon.*
One more extension is the development of alternative (partial) solutions and their assessment for decision support during a configuration. This will be interesting for online configuration, too, because there is no expert available to influence the customer in his decision. Configurations have to fit different heterogeneous requirements which can lead to various conflict assessments. There is a conflict resolution framework described in [Hollmann et al., 2000] which can be integrated in *EngCon* for multicriteria evaluation support.

REFERENCES

[Arlt et al., 1999] V. Arlt, A. Guenter, O. Hollmann, L. Hotz, T. Wagner. *Engineering & Configuration - a knowledge-based software tool for complex configuration tasks*, in AAAI-99 Proceedings, Orlando, AAAI-Press 1999.

[Brinkop, 1999] A. Brinkop. *Variantenkonstruktion durch Auswertung der Abhaengigkeiten zwischen den Konstruktionsbauteilen*, Infix, 1999.

[Boehm et al., 1999] A. Boehm, H.J. Mueller, J. Rahmer, S. Uellner. *A Discussion of Internet Configuration Systems*, in AAAI-99 Proceedings, Orlando, AAAI-Press 1999.

[Guenter, 1992] A. Guenter. *Flexible Kontrolle in Expertensystemen zur Planung und Konfigurierung in technischen Dom"anen*, Infix, 1992.

[Guenter, 1995] A. Guenter. *Wissensbasiertes Konfigurieren: Ergebnisse aus dem Projekt PROKON*, Infix, 1995.

[Guenter and Kuehn, 1999] A. Guenter and C. Kuehn. *Knowledge-Based Configuration - Survey and Future Directions*, in XPS-99 Proceedings, Lecture Notes in Artificial Intelligence No. 1570, Springer-Verlag, Wuerzburg, 1999.

[Hollmann et al., 2000] O. Hollmann, K.C. Ranze, H.J. Mueller and O. Herzog. *Conflict Resolution in Distributed Assessment Situations*, in H.J. Mueller and R. Dieng (eds.), *Computational Conflicts - Conflict Modeling for Distributed Intelligent Systems*, Springer-Verlag, 2000.

[Sabin, 1998] D. Sabin, R. Weigel. *Product Configuration Frameworks - A Survey*, in IEEE Intelligent Systems July/August 1998, Seite 42-49.

[5] http://www.bme.de/bmecat
[6] http://www.xmledi.org