4th Int. Configuration Workshop Proceedings (associated with ECAI 2002 conference Lyon) - Chaired by Michel Aldanondo, Toulouse University - Mines Albi - France

Michel Aldanondo
École des Mines d'Albi-Carmaux

Configuration

Papers from the Workshop at ECAI 2002
Workshop n°4
Michel Aldanondo, Chair

22-23 July 2002
15th European Conference on Artificial Intelligence

University Claude Bernard of Lyon 1 (UCBL)
National Institute of Applied Sciences (INSA)
Lyon - France

ECAI 2002 Configuration Workshop

Special thanks
Paul Gaborit, Ecole des Mines d'Albi Carmaux, France.

ECAI 2002 Configuration Workshop

Contents

ECAI 2002 Configuration Workshop

Preface

Configuration tasks can be defined as designing a product individual using a set of pre-defined component types while taking into account a set of well-defined restrictions on how the component types can be combined. Product configuration problems are present in many domains: manufactured product (machine, computer, furniture…), networks (telecommunication, transportation systems…), service (banking, travel…) and software (ERP, CRM…). Computer programs called configurators, using AI techniques such as constraint satisfaction, its extensions, description logic, logic programs, rule-based systems and different specialized problem solving methods, can support it.

The ECAI 2002 workshop on configuration is the fifth in a series of workshops started at the AAAI 1996 Fall Symposium and continued at the AAAI 1999 Orlando USA, ECAI 2000 Berlin Germany and IJCAI 2001 Seattle USA. The goals of these workshops are to promote high-quality research in configuration and strengthen the interaction between industry and academia. The industrial interest in the field is indicated by the increasing number of configurator vendors. The importance of configuration has expanded as more companies use configurators to efficiently customize their products for individual customers. Combined with e-business solutions they have a high market impact and generate new business opportunities in many industries for new products and new ways to interact with customers.

However, efficient development and maintenance of configurators require sophisticated software development techniques. AI methods, more than ever, are central to designing powerful configuration tools and to extending the functionality of configurators. The working notes of this workshop gather contributions dealing with various topics closely related with configuration problem modeling and solving. The 25 papers demonstrate both the wide range of applicable AI techniques and the diversity of the problems and issues that need to be studied and solved to construct and adopt effective configurators.

Michel Aldanondo
Ecole des Mines d'Albi Carmaux

E-Configuration Enterprise
A New Generation of Product Configuration System

Bei Yu *

1 CONFIGURATION TECHNOLOGY

Configuration technology (see Figure 1) is first and foremost the underlying concept for representing a product or a service that has features or parts to be configured. This structured representation is called a *product model*. Configuration technology is the modeling language and maintenance environments that simplify the creation and maintenance of a product model.

Second, configuration technology is the runtime inference engine/solver that enables the end user to make correct configurations based on the model in an ingenious and intuitive fashion.

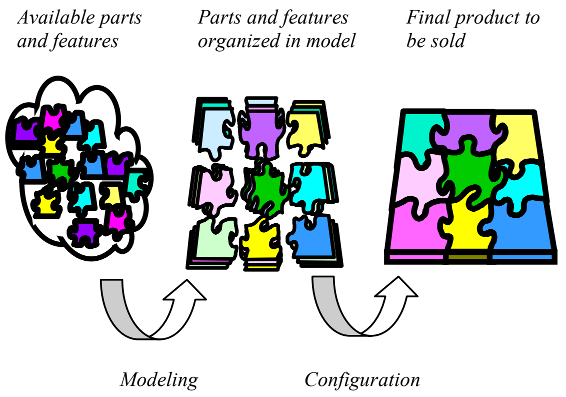

Available parts and features
Parts and features organized in model
Final product to be sold
Modeling
Configuration

Figure 1. Visualization of the Modeling and Configuration processes.

The overall strategy is to produce 100% shrink-wrapped software for a large variety of configuration problems, mainly in CRM applications using the same modeling methodology.

This paper describes a new approach to product configuration methodology.

2 THE BUSINESS CHALLENGE

One of the most critical challenges for companies that implement configuration and web-enabled products is to address both the usability and maintenance issues at the same time.

When talking configuration as part of the business processes there are three main issues to consider:

1. Can I implement a configurator that can be **maintained** in a way that fits my organizational structure, without having the update process to be a bottleneck?
2. Can I deploy a solution that supports customers needs and my general business needs (interactive and fast response time) - and will it be able to scale to fit future needs?
3. Will the tool provide **100% correct** configuration - always?

----

*Baan CRM, Hørkær 12A, DK-2730 Herlev, Denmark.
E-mail: byu@baan.com

The need is to have a configurator that combines more ways of stating relations, calculations and constraints into one modeling environment with infinite solution space. It should be 100% declarative allowing for the handling of very big and complex problems without programming. This is the only viable way for the product to stay maintainable when handling more complex configuration issues.

Supporting general business needs in today's environment encompass supporting the globalization both from a modeling and a deployment point of view. Specific points that support the globalization are object-oriented modeling, concurrent modeling, table-based constraints, and separate UI data.

In the configuration process, the system guides the user to the selections that can be made in order to get a solution at the end – based on the principle of "complete deduction". This allows an interactive real-time guided configuration processes – with no rework and a deliverable solution. For the customer or the sales rep this means instantaneous feedback on any selection made, even in complex product and pricing environments. This accurate overview and guidance is particularly important when a customer is ordering in an unassisted mode e.g. via the web.

3 MASS CUSTOMISATION

The philosophy around mass customization is to sell customized solutions to customers who are willing to pay extra to get a custom-made order. At the same time, the batch size in the production process shall be optimized, by producing standard items and assembling these components to make the unique product for the customer. By using this business philosophy the companies can optimize the turnover and at the same time minimize the production cost – maximizing the profit – to the benefit of both the company and the customers.

This means that some Engineer-to-order companies can move into made-to-order, and thereby minimize production costs. Also, some standard-components manufactures can offer their customers special solutions, moving into assemble-to-order to maximize their turnover.

The configuration component is a very important part of this Mass Customization philosophy, where the customer needs can be linked to the available offerings - and thereby generate the foundation for the fulfillment process (Customer needs -> List of components + component dependencies -> Production routes/schedule -> Delivering the products).

4 E-CONFIGURATION ENTERPRISE

E-Configuration Enterprise is the solution for configuration problems in real life. It fulfills the user interaction requirements given by [1]. E-Configuration Enterprise, as the new generation of our configuration system, has unique features compared with the previous systems [2], such as:

1. A brand new configuration modeling language;
2. Object-oriented modeling approach;
3. More intuitive product description/modeling;
4. Wide-range covered constraint expressions;
5. More powerful inference engine.

E-Configuration Enterprise consists of product modeling environment (Modeler) and configuration environment (Configurator) both PC-based and Web-based. In the Modeler, the user can make a product model and a User Interface (UI) model, and tables, if necessary. These are wrapped into a package, which can be loaded into the Configurator for configuration processing(see Figure 2).

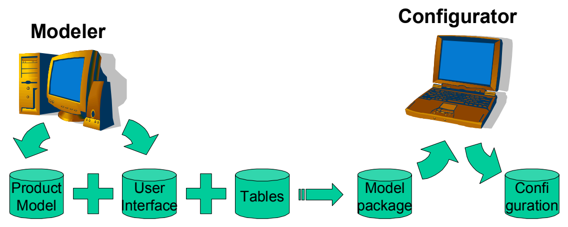


Figure 2. E-Configuration Enterprise product configuration concept. The Model Package holds all information generated by the Modeler and used by the Configurator.

This section only highlights the system kernel features. The user-friendly interface for modeling and configuration could be shown through the demo session.

4.1 CAVA

To encompass all configuration problems - especially those that are very large and dynamic - a new modeling technology called CAVA has been designed and constructed. CAVA is a configuration modeling language. The CAVA language is aiming at a standard configuration modeling language that is able to encompass all configuration problem domains [3]. This unique language is certainly the first step towards the ambitious goal. It is able to describe almost all configuration problems within very complex products in large companies.

The CAVA modeling language is declarative and constraints based. The expressiveness of constraints is able to perform reasoning about identifiers, booleans, integers, reals, strings, databases, and functions.

Typically, different configuration domains require different tools and technologies. Today's commercial applications are strongly focused on tools and technologies that address the specific problems of these specific configuration domains [3]. For a growing number of businesses, this is not where the added value of configuration is found. CAVA outlines the foundation for common modeling approach through all applications.

The main issue with all configuration solutions is to get enough expressiveness so that the real world products can be expressed in a model. However, there is a tradeoff issue with all existing methodologies, which either makes it easy to maintain and limit the expressiveness or have a large expressiveness but nevertheless will make the maintenance a hard task.

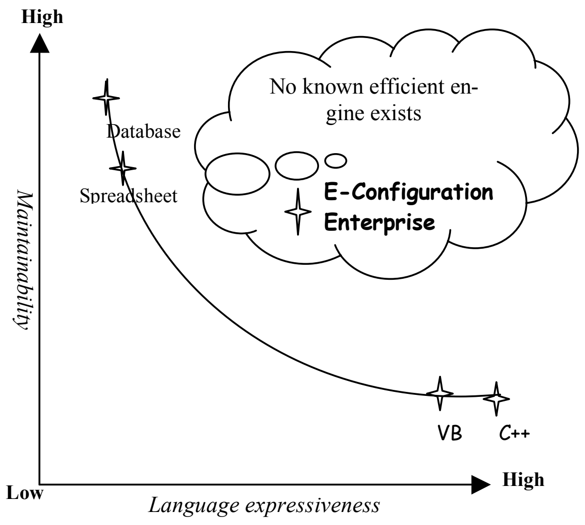


Figure 3. Compromised position E-Configuration Enterprise

CAVA is a compromise between maintainability and language expressiveness (see Figure 3). CAVA sets out to cross the boarderline that has prevented tools from overcoming this tradeoff. The tradeoff is mainly due to limitations on the computational aspect of configuration problems namely that they by nature are very difficult (NP-Hard). However, recent progress within new algorithms and hardware performance has made it possible, combined with a set of unique patented technology, to cross this line.

4.2 Object-oriented modeling

Object-oriented modeling means that the product model data is made of components (classes), which can be reused by having several copies (instances) of this component (class) in another component (class). For example, in a bookcase (a class) there can be a number of shelves (copies of a shelf component class). In the product model, the shelf component is only defined once, and then used in the bookcase component *n* times.

Another concept in object-oriented modeling is “inheritance”. This means that a component (class) can inherit features from another class so that it appears as if the class in fact has these features. This is a very powerful feature when creating a product family hierarchy. For example, a bookcase can be either a classic bookcase or a modern bookcase, where some special features are added to the specific kind of bookcase, while still having the basic bookcase components defined only once. The inheritance principle is very useful in order to reduce model data redundancy and thereby increase the maintainability of the model.

4.3 Table based constraints

The product model can use data in relational tables/databases to define product feature relationships. These tables can be either internal (defined in the model) or external (defined in an external database), from where data is read by the Configurator. This will allow the owner of the product model to have product relations/constraints defined in a database that can be applied to the user environment without having to re-compile the product model.

From a maintenance perspective, this is a good example for CAVA to show how easy it is to maintain constraints. Rather than writing hundreds of constraints, a table structure with one binding will make these constraints redundant.

4.4 Hard and soft constraints

Hard constraints are relations that have to be fulfilled in order to have a consistent model and/or configuration, where soft constraints are relations that can be violated or changed. Soft constraints are prioritized so that it is possible to apply different soft constraints for the same feature. A soft constraint can be used as pre-defined defaults. For example, when a user selects the color of a car, the color of the interior is set to match the color of the car. However, this can be overruled if the user wants another color.

4.5 Support for concurrent modeling

The product model, including the UI, can be maintained by a number of people, where the ownership for the different parts of the model is assigned to more than one person.

The advantage is to be able to maintain parts of a model without risk of overwriting other changes made at the same time in other parts of the model.

4.6 Separation of product data and UI

The product data describes the features and options and the relations between them, where the UI data describes how the product data is being shown to the user at run time. By separating the product data and the UI data, it is possible to make several UIs for the same product data. This can be beneficial if the same product data is used in different environments, for example, a UI for the direct sales force and another UI for the partners/dealers in an indirect channel.

4.7 Web standard based and Web-enabling

When the UI is created, a UI Builder transforms the user interface definitions into HTML. If needed the HTML page can then be modified using any HTML editor.

The integration is based on industry standard XML both when exchanging model information and when exchanging data.

E-Configuration Enterprise is a web-based thin client product, and the only application needed to make a configuration is a web-browser. On the server side, using the extensive XML-interface or the COM-interface can make integration to business solutions.

4.8 Activations - user and situation dependent models/UI

Activations can be used to enable or disable features and constraints. These activations can be based on time (Time activations), for example price offer period, or situation (Group activations), such as country availability. This can be used to hide or deselect features that some users are not allowed to see, or features that are unavailable at the time of configuration. Hiding means that the option does not show up in the UI, where deselect means that this option is not available (dimmed at the UI).

4.9 Trace and Alternatives

The Trace mechanism is supposed to help a modeler to understand the “why” questions, such as, why a value was moved from a variable; why a variable was bound to a value; why an instance was created as well as why a contradiction exist. This information is intended to assist the model engineer during modeling.

The Alternatives is a service which 1) helps the end-user to make a selection that otherwise would lead to a contradiction, and 2) helps the end-user solve a contradiction.

The information given to the end-user is the same in both cases, i.e., user selections that need to be released.

4.10 Scalability for large models

Due to the principle of “Field of Interest” the Configurator has good runtime response time and performance for large models.

The Field of Interest concept means that the Configurator will first verify or deduct selections in the pages or instances that are shown to the user, and second verify or deduct on the rest of the model. In this way, the Solver can maximize the support given to the user should it be interrupted before it has searched all features in the model.

4.11 Complete deduction

Complete deduction in the Solver/Inference Engine is based on a patented method [5], which is the foundation for the uniqueness of our configuration technology. Complete deduction allows the people doing the modeling to focus on individual constraints for each object in the model. The system will then secure the complete solution overview, i.e. make sure that the model is consistent. This brings down both implementation and maintenance times by reducing the number of constraints you have to implement by a factor of 10 to100, compared to other technologies.

4.12 Deduction levels

In the Configurator, the user can set maximum response time before the Solver is interrupted and the result is presented to the user. Depending on the size and complexity of the model the Solver will not have time to reach the highest level of accuracy, but will stop at some lower level at timeout.

There are six deduction levels that the solver can reach: *Accepted*, *Propagate*, *Look-ahead*, *Solution*, *Unbound,* and *Complete*. The Solver works sequentially at one level at a time starting with Propagate and ending with Complete. This means that if a certain level has been achieved then the deduction levels below have also been achieved.

Because of the nature of reals however, using reals in the model does not necessarily remove every invalid value from the model.

4.13 Infinite solution space

The user can dynamically add components and/or parts at run time, and the selections around these are validated as an integrated part of the product model. For example, the users can at run time select the number of shelves in a bookcase, and subsequently configure each shelf individually. This dynamic extension of the components in the solution often leads to a multilevel configuration solution.

4.14 Integration to entire business solution

E-Configuration Enterprise, as one key component, has been integrated into company business solution offerings. Configuration issues nowadays cannot be considered independently, but will have to relate to other business issues in a company. Such integration capabilities make the configuration system more powerful in terms of solving all configuration problems across whole business processes from engineering, manufacturing to sales and marketing.

5 SUMMARY

Issues like usability, maintainability, and optimal time to market are often considered contradictions when the business solution involves a configuration component. The demand for providing fast, correct, and complete information to customers and channel partners while keeping costs under control is just one of the contradictions. If companies want to effectively solve this dilemma, they need to be using new technologies to solve these problems.

E-Configuration Enterprise contains a unique and patented technology that effectively addresses all issues of configuration. The separation of the product data and the UI makes it possible to link the configuration model to other user interface components, including third party web controls, that address the UI needs for the company.

E-Configuration Enterprise is a compelling software component that easily integrates into business solutions.

The open architecture will provide integration with web-based solutions, as well as APIs for other integration purposes.

E-Configuration Enterprise is the first step into a new world of configuration, while not burning any bridges to previous solutions.

ACKNOWLEDGEMENTS

Although I am solely responsible for the paper, all developers who are involved in the project contributed to the success of the E-Configuration Enterprise. I would also like to thank Mette Nyberg and Beverly Poulsen from the Documentation department for their help with formatting and reviewing the paper.

REFERENCES

[1] F. Frayman, User-interaction Requirements and its implications for efficient implementations of interactive constraint satisfaction systems, Workshop Proceedings on User-Interaction in Constraint Satisfaction, International Conference on Constraint Programming and Logic Programming (ICLP’01), p31-41, 2001.

[2] B. Yu and H.J.Skovgaard, A configuration tool to increase product competitiveness, IEEE Intelligent Systems and Their Applications. July/August 1998.

[3] S. Mittal and F. Frayman, Towards a generic model of configuration tasks, in Proceeding of the 11th International Joint Conference on Artificial Intelligence (IJCAI), 1395-1401, 1989.

[4] D. Sabin and R. Weigel, Product configuration frameworks – a survey. IEEE Intelligent Systems and Their Applications, July/August 1998.

[5] G. Møller, Signal Processing Apparatus and Method. Patent WO 90/9001.

Configuring Software Products with Traditional Methods – Case Linux Familiar

Katariina Ylinen[1], **Tomi Männistö**[1] and **Timo Soininen**[1]

Abstract. Recently, the software industry has begun to adopt a product family approach. Support tools for tailoring configurable product families have been developed for some time for mechanical and electronics products. However, it is not clear that the modeling languages designed for these configurators are suitable for software product configuration. In this paper, we investigate this problem by modeling a Linux Familiar operating system distribution with a configuration modeling language aimed at representing the structure of physical products. Findings from this case study suggest that the language is largely suitable for software product configuration. However, some phenomena in the product strongly suggest that modeling them as functions, features or resources and optimality criteria familiar from the configuration domain would be useful. In addition, there is some evidence that deeper modeling of versions of components and reconfiguration knowledge, not usually covered by configuration models, should be supported.

1 INTRODUCTION

Configurable product families have been an important phenomenon in the mechanical and electronics product domains for some time. In these domains, called *traditional products* for short, it has been noted that it is possible to satisfy a wide range of customer requirements with relatively low costs by designing the products to be routinely configurable. Systems that support tailoring such products, *configurators*, have been extensively developed and studied [4]. Recently, the software industry has to some extent adopted the point of view that designing software product families and delivering variations of them rather than customizing individual products can be far more cost-effective [1]. However, it is not clear that configurators supporting traditional products are suitable for software product configuration. The software community has approached product families from a different perspective and developed its own modeling methods and methods for tailoring the products [15,16] .

In this paper, we investigate whether there are any differences between the two domains of traditional products and software by means of a concrete case study. We try to model a Linux Familiar operating system distribution [13] with a configuration modeling language aimed at representing the structure of physical products [11, 14]. The three research questions we ask are: Is the configuration modeling language adequate for representing the Familiar configuration knowledge? If not, what are the most important missing elements? And finally, which of the common concepts used for modeling configurable traditional products [5] could be used to make the models more useful?

The rest of the paper is organized as follows: We first briefly review and compare the two domains of product configuration and software families and analyze their similarities and differences in section 2. We then present the case product and the related configuration problem and present an analysis of the configuration knowledge for the product in section 3. We present a way to model the configuration knowledge of the case product using a language for modeling traditional products in section 4. The findings of our case study are presented and discussed in section 5. Finally, in section 6 we present our conclusions and identify some topics for further research.

2 BACKGROUND

This section briefly describes the domains of product configuration in both traditional industry and in the software industry and then compares the frameworks used in them to find their similarities and differences.

2.1 Physical Product Configuration

For a *configurable product family,* each *product individual* is adapted to the *requirements* of a particular customer order on the basis of a predefined *configuration model,* which describes the set of legal *product variants* [6,7]. A *configuration*, that is, a specification of a product individual is produced from the configuration model and particular customer requirements in a *configuration task.*

Knowledge based systems for configuration tasks, product *configurators,* are an important application area of artificial intelligence techniques for companies selling products adapted to customer needs [9,10]. Product configuration tasks and configurators have been investigated for at least two decades [11].

Conceptualization of configuration knowledge synthesizing these approaches is reported in detail in [7,17].

2.2 Software Product Configuration

The software professionals often claim that the methods and processes of traditional industry do not apply to software development and products. It has been suggested that the product configuration concepts and methods used for the physical products are not directly applicable in the software industry [2]. According to Brooks [6], software has four special features that make it special: complexity, conformity, changeability and invisibility.

Even though modern physical products may also be complex and invisible in nature, they are constructed of a limited number of parts that are replicated. In a software product variant, there are usually no two parts equal to another.

In the past, much of the configuration-related software management research has focused on the software configuration management (SCM). In SCM, the focus has primarily, although not completely, been on managing the evolution of the source code files

[1] Helsinki University of Technology, Software Business and Engineering Institute SoberIT, P.O. Box 9600, FIN-02015 HUT, Finland. Email: {Katariina.Ylinen, Tomi.Mannisto, Timo.Soininen}@hut.fi

[7]. Most SCM systems today only manage the software systems as a set of files instead of as configurable product components [7].

The research of configuring final software products has only recently become an increasingly important challenge while the industry has slowly adopted a goal of producing product families instead of single products. A related field, which aims at modeling product families, is the research on architectural descriptions [17]. Even though the primary goal of ADLs has not been the product configuration task, it can be noted that they have adopted significantly similar concepts as the traditional configuration research.

While formal methods for software product configuration have been absent, there has been separate efforts for configuring single products. Most of the software companies implement configurability by producing one big product with all variants included. The configuration task is then done using, for example, preprocessor directives or makefiles to define the specific product variant [2].

Another result of software invisibility is that in software products, it is often impossible to know the correct order of installation, if there are limitations to this. This may often be as complex in the physical products as well. The difference lays in the usage – software products are often installed at the same time as they are configured. The installation and configuration process is, especially in consumer products, assumed to happen at the same time and at least semi-automatically. This has led to a situation where installation programs do the configuration and vice versa.

The adequacy of the configuration knowledge present in the packages of Debian distribution has been a subject for study before [3, 12]. The approach has been developing a new rule-based language for modeling the packages and dependencies, and then implementing this using a logic-program-like rule based system. We, on the other hand, aim at applying the high-level modeling concepts of physical product configuration to modeling Linux Familiar.

3 FAMILIAR CONFIGURATION PROBLEM

Linux Familiar was chosen as the case since it is developed for a handheld computer, Compaq iPAQ, and therefore faces resource limitations, such as the amount of memory available. Whereas operating systems on PCs can be installed with all bells and whistles included, the handheld devices make prioritization of installed components necessary. There are currently about 1700 Familiar packages available in the distribution, only a subset of which fits into a device at the same time. The packages vary in nature from necessary Linux kernel packages to application packages. Linux was chosen instead of other handheld operating systems due to its openness, as it was easier to study.

Linux is a monolithic operating system. The part implementing the core functionality of the operating system such as the thread management, file system etc. is called the Linux kernel [9]. The kernel itself is an interesting entity from the configuration perspective, since it has built-in support for dynamic reconfiguration. However, this feature has been implemented for mainly memory management use and is left out of the scope of this paper. We focus our attention on package management, that is, a higher-level configuration management of the whole distribution.

The Familiar Distribution is derived from the Debian distribution, which means the configuration attributes share the same concepts. Both distributions consist of a large amount of software components, called packages. Packages consist of several files, which can be either executables or other files as well. In Debian, as well as in most widespread Linux distributions, there is a package management system, which manages the installation and removal of packages in the system. In Debian, this program is called dpkg and it manages configuration constraints like dependencies and conflicts between packages, virtual packages and installation order. It can also be used to create and purge packages.

Due to the size constraints dpkg is not a part of Familiar but a lighter version of it called ipkg has been developed. Ipkg shares the basic functionality in package installation and removal but lacks most of the configuration validity functions. Therefore, the comparison with the traditional configurator is made against dpkg features. Familiar packages are equipped with configuration information of the same syntax, and they can still be used as an example of the input data.

The configuration language of Familiar / Debian is fairly simple. It is a list of the package information in pure text format. There are two fields attached to all packages: version and the package name. In addition, it lists all constraints known for the package. The constraint clauses also refer to a package name and optionally to its version. An example of the fields concerning configuration in a Familiar package description:

```
Package: xlibs
Priority: optional
Version: 4.0.2-13
Replaces: xlib, xbase (<< 3.3.2.3a-2), xlib6 (<< 3.3.2.3-2)
Provides: libxpm4
Depends: xfree86-common (>> 4.0), libc6 (>= 2.2.1-2)
Conflicts: xlib, xlib6 (<< 3.3.2.3-2), xlib6g (<< 4.0)
```

Figure 1

The different types of these packages is explained in more detail in section 3.1. The nature of the different relationships between the packages is analyzed in section 3.2.

3.1 Package Types

Currently, three types of packages can be identified in the system. The package types are *virtual, concrete* and *task* packages.

A virtual package is an abstract package that does not actually exist and, the functionality of which can be provided with one or more concrete packages. The virtual packages do not appear on the package lists as package definitions and therefore cannot have relationships to other packages. Instead, some of the concrete packages refer to a virtual package as they provide its functionality.

In Debian, the developer community strictly controls the virtual package names and the full list of available virtual packages can be found at the developer web site [10]. The same virtual package names are used in Familiar, although it seems that the control is a bit lighter and there are some additional, non-documented ones as well. An example of a typical definition of virtual package (x-terminal-emulator) in a Familiar package description looks like this:

The concrete packages are the actual packages that can be installed or uninstalled on the device and that have relationships to other packages on the system. The concrete packages consist of one or more files to be saved on the file system. These files can be either executables or other files, for example text files.

The third package type, task package, is a package that consists of several other packages. It has no own separate functionality but just the collection of the packages it contains. The package does not withhold any important functionality itself but has several dependencies to other packages, so that installing the package requires then installing the whole set. The task packages seem to consist of a set of different packages that together implement certain functionality. The task packages in Familiar are separated from concrete packages with a naming convention. The package names have prefix "task-". The task packages are made to make basic installations easier for the end user by collecting a potentially important set of related packages into one package.

3.2 Relationships

As mentioned earlier, the package descriptions include constraint clauses. Every package has a listing of these as presented in example in figure 1. The clause first describes the nature of the relationship and then a list of package names and their versions.

There are seven different kinds of constraint types:

- Depends <*package B*> - this package requires package B to be installed on the device to function correctly.
- Pre-depends <*package B*> - it is required that package B is installed on the device before this package can be installed.
- Conflicts <*package B*> - this package should not be present in the same configuration with package B.
- Provides <*package B*> - this package provides all the functionality and files present in package B.
- Replaces <*package B*> - the installation of this package removes or overwrites files of package B.
- Recommends <*package B*> - the package B is recommended when it is presumable that the users would like to have it in the configuration with this package.
- Suggests <*package B*> - the package B is suggested to get better use of the package.

Of these, the constraints recommends and suggests are not relevant for checking the configuration validity but only useful hints for the user for configuration optimization. It can be also noted that the *pre*-depends clause is used to gain correct installation order but does not differ from depends when used to check the configuration validity.

For all the different relationships, there can be many packages listed and in that case the relationship holds for all the packages in the list. That is, the commas between the package names can be interpreted as Boolean AND. For depends, there can also be a Boolean OR, which is indicated by " | " in the list. The precedence rules are not very clear and we are not sure for which elements the OR statement refers to in some occasions. It is not, however, a big problem since there are not many OR statements currently in the package descriptions.

Depends is a simple relationship. When a package depends on another, the other must also be installed on the system. When checking the configuration validity, pre-depends is also treated the same way.

There are also some interesting exceptions in the use of "Depends" relation, which we consider more or less misuse and not usage rules. For example, the package *xlibs* depends on its own older version. This means there are also some incremental packages – the newer version is not actually a new version of the package but some additional features for it.

Conflicts is just as simple. When a package is in conflict with another, they should not be installed on the same system. There is, however, a minor problem in the language and the configurator, dpkg, considering this. As a package is installed in the system, only the relationships of the installed package are checked. In case there is a package in the system conflicting this new package, it may not get noticed. This is due to the fact that as the conflict has been noticed, it is highly probable that it only has been declared in the description of one of the packages but not in both. This means that either the conflict information must be duplicated to both of the package descriptions or the functionality of dpkg should be changed so that it would check all package descriptions of the packages installed on the system to check the configuration validity.

Provides has been designed for managing packages, which in some way implement functionality available in some other package. Unlike the other configuration clauses, the provides has been used quite systematically but it has two different tasks. One of them is the usage for version control when package naming has changed during versions. The newer version provides the older one, and it seems that the potential dependencies from other packages to the older version will not get broken. Provides allows the coexistence of the provided and the providing package in a configuration. In the case of renaming an existing package when making a new version, it must be separately specified with a conflict clause that the versions should not be present in a configuration at the same time.

Provides is also used for virtual packages – several concrete packages can provide some virtual package functionality, and many of them can coexist on the configuration [3]. For example, a concrete package *rxvt*, which is an emulator program emulating the x-terminal, has a row: "Provides: x-terminal-emulator" in its package definition. The *x-terminal-emulator* is a virtual package, and packages like *rxvt* (a basic x-terminal emulator) or *rxvt-aa* (an x-terminal emulator with anti-aliased fonts support) could coexist on the system both implementing the virtual x-terminal emulator package.

An example of the virtual package use of Provides in Familiar package description is presented in picture 2.

```
Package: rxvt
Version: 1:2.6.3-8-fam6
Provides: x-terminal-emulator
Depends: libc6 (>= 2.1.97), xlibs (>= 4.0.1-11)
Conflicts: suidmanager (<< 0.50)

Package: rxvt-aa
Version: 1:2.6.3-8-fam6
Provides: x-terminal-emulator
Depends: libc6 (>= 2.1.97), xlibs (>= 4.0.1-11), libxft, libxrender
Conflicts: suidmanager (<< 0.50)
```
Figure 2

Replaces is quite ambiguous and is therefore used in various ways, some of which can be clearly interpreted as designer errors. It is also the most dangerous one in use, since there is no such system functionality that would remove the replaced package beforehand. As there is no proper information on which way the replacement exactly takes place, there is no guarantee on what will happen to the files installed originally with the package being replaced. In practice, the replacement is therefore only used in some cases when user tries to install two conflicting packages. In these cases, when one of these packages is marked as replacing the other, the installer in Debian prefers the replacing one [3].

Replaces is used, as already stated, very differently in different packages. In some packages, it is used similarly as provision clause. As the replacing package should replace, by definition, some of the files of the replaced package, it is also dangerous that there is no solution so far for the case when a user would try installing the replaced package back to the system. The replacement clause is again only written in the other package and thus overwriting the same files back again would most probably break the system. This may be one reason for the fact that the replacement has not been often used in actual replacing of packages. One example of an unexplainable use of this relationship is the package set util-linux (2.11b-2-fam2), fileutils (4.0.43-1) and shellutils (2.0.11-5). Both fileutils and shellutils replace an older version of util-linux. Still, they seem not to provide the functionality specified in the util-linux description. There are no other relationships between these packages.

We conclude that the most common use for replacement is that the replacing package provides at least some of the functionality of the replaced one, but it is not promised that it works the same way. This means the dependencies from other packages to the replaced package will get broken when it gets replaced, unlike in the case of provision. This cannot, however, be generalized for all packages due to the very varying use of the concept.

4 FAMILIAR AFTER MAPPING

In this section, the new configuration model for Linux Familiar is introduced. The syntax of the used language is presented in section 4.1 and the mapping of components and relationships are presented in sections 4.2 and 4.3, respectively.

4.1 Modeling Language

We use a language based on a subset of a general configuration ontology [5] to try to model the Familiar configuration information to gain understanding of how suitable the concepts of the language are for modeling software.

The used language, called *PCML,* is introduced in [11]. The main concepts of PCML are *component types*, their *compositional structure*, *properties* of components, and *constraints*. Component types define intensionally the characteristics (such as parts) of their *individuals* that can appear in a configuration. A component type is either *abstract* or *concrete*. Only an individual directly of a concrete type is specific enough to be used in an unambiguous configuration. 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*. A component type may define properties that parameterize or otherwise characterize the type. A *property definition* consists of a *property name*, a *property value type* and *a necessity definition.* Component types are organized in a *taxonomy* or class hierarchy where a *subtype* inherits the property and part definitions of its *supertypes* in the usual manner.

Constraints associated with component types define conditions that a correct configuration must satisfy. The first level building blocks of the constraint language are *references* to access parts and properties of components, and constants such as integers. References can be used in succession, e.g. to access a property of a part. Boolean returning *tests* are constructed out of references, constants, and arithmetic expressions. Tests also include predicates that allow checking if a particular referenced individual exists or is of a given type. Property references can be used with constants in arithmetic expressions that can be compared with the usual relational operators to create a test. Test can be further combined into arbitrarily complex *Boolean expressions* using the standard Boolean connectives.

4.2 Components

The component types we define for Linux Familiar configuration are the package types we identified in section 3.1. In addition, we define a component type defining the whole system, of which the packages are parts. We define a type hierarchy, in which there is one supertype *package*, of which all packages are subtypes. Package is an abstract component type with one property, version, which is of value type string. Both concrete and virtual packages are subtypes of this component type. It is a subtype of the root component type of the language, Component.

The virtual component types are defined as abstract, as there should be no occurrences of component individuals of them in a valid configuration. The conceptual meaning of the abstract component is seen as the same as that of the virtual package. All the virtual packages are modeled as subtypes of the Package component type. The concrete packages implementing a virtual package, are subtypes of the virtual package in question. Other concrete packages will all be subtypes of the component type Package.

Some of the data on the packages that is needed on configuration time are modeled as properties for concrete packages. These are priority, size, installed-size, and version. Priority will be modeled with the concept of cardinality. When a package is obligatory (priority: required), its cardinality is 1 and when optional, it is 0 to 1. Virtual packages will not have these properties since this information is not available for them.

4.3 Relationships

Relationships between the packages are modeled as constraints in the configuration model. The relationships are translated as follows:

A Depends B	constraint <constraint_name> A implies B
A Pre-Depends B	constraint <constraint_name> A implies B
A Conflicts B	constraint < constraint_name> not (A and B)
A Provides B (virtual packages)	subtyping, A is a subtype of an abstract component type B
A Provides B (change of name between versions)	subtyping and conflict, A is a subtype of B and constraint <constraint_name> not (A and B)

As many of the relationships also involve version numbers, they must be taken into account in the constraints. Constraints involving version numbers are modeled as the following example demonstrates (conflict):

```
A Conflicts B (>= 2.2.1)   ⇨
constraint <constraint name>
not(A and B and
<part name for B>.B:version >= "2.2.1")
```

4.4 A Sample

Figure 3 presents a sample of the model created with PCML:

```
configuration model TestModel

 # The concrete package type
 component type Package
  abstract
  subtype of component
     property version value type string

 # The virtual package x_terminal_emulator
 component type x_terminal_emulator
  abstract
  subtype of Package
```

```
# Package rxvt implementing x-terminal-emulator
 component type rxvt
  subtype of x_terminal_emulator
     property version
        value type string constrained by $
        in list ("1:2.6.1", "1:2.6.3-8-fam6")
  . . .
component type OSystem

  part x_terminal_emulator
    allowed types x_terminal_emulator
    cardinality 0 to 1

  constraint only_one_version_rxvt
    present(x_terminal_emulator) and
    x_terminal_emulator individual of rxvt
    and x_terminal_emulator.rxvt:version
        = "1:2.6.1"
```
Figure 3

In this example, we use the same virtual package as in figure 2, ‘x_terminal_emulator. Two concrete packages, rxvt and rxvt-aa, implement its functionality. There are two versions of each available, but for a valid configuration, only the older version of rxvt is accepted.

5 DISCUSSION AND FUTURE WORK

Linux Familiar does not represent the configuration aspects of all software products. However, it is an easily studied real-world example of configurable software. On the basis of the modeling approach defined earlier, it seems that the methods of physical product configuration are at least partially applicable to software products as well. The differences between these domains are not so remarkable that entirely new methods and modeling languages need to be developed for software configuration.

Using a modeling language like PCML seemed adequate for modeling the most important aspects of Linux configuration. The ability to model virtual packages as entities on their own with relationships to one another can be considered a good additional feature, as long as we have correctly understood the concept. If, however, the virtual package concept has been developed for describing the functionality offered by different packages, modeling them as features or functions [5] could be more useful. Those concepts correspond semantically better to what seems to be modeled with virtual packages and provide more flexibility in defining relationships between virtual and concrete packages. For now, as each virtual package is implemented by one package only (although there are several alternatives), it can be stated that the use of abstract component types and subtyping is a reasonable choice. This should, however, be studied further.

On the basis of this case study, there is no indication that the differences between the traditional industry and software domains, referred to by Brooks [6], are of big relevance with respect to product configuration. Invisibility, conformity and complexity did not have an impact in the case product. On the basis of this single case study, it seems that changeability may be more important and a difference between the domains. A single software component can easily have quite a large number of versions. However, Linux is somewhat a special case in this respect as more functional versions are released for users than in a typical commercial software product. Therefore, it should be studied further if the number of versions really is big for software products in general and what sort of challenges this may present for the modeling task. In our case, there were no complex and large version spaces, which means that no conclusions could be done of the issue.

Modeling conflicts was easier using PCML. For example, the modeling of conflicts was simplified. The model then became more manageable as the problems presented in section 3.2 ceased to exist.

The usefulness of cardinality of packages in software configuration was questioned when the mapping of the concepts was made. In this case, there were only optional and required packages and therefore a concept of optionality could be used instead of cardinalities. On the other hand, modeling distributed systems may require a more elaborate cardinality, as a component type may be instantiated (i.e., installed) on several different devices. Thus, we cannot conclude that cardinality would be useless for software configuration modeling.

In the case of a handheld device, a concept for modeling the configuration size would have been useful. In PCML, Such a concept is not present and some problems can be expected when the disk of the device becomes full. The size information was available for all the packages but there were no means used to calculate the configuration size in this case study. However, the resource balancing based approach to modeling incorporated into the ontology of Soininen et al [5] could be useful to capture this phenomenon.

5.1 Problems and Challenges Raised in Our Mapping

We faced a few challenges when modeling Linux Familiar package descriptions with PCML. We next discuss the challenges with the modeling and the challenges with the input data in more detail.

Dangling references and reconfiguration. In the package descriptions of Linux Familiar, there were a remarkable number of references to packages, which were not present in the package list. PCML does no allow references to non-existent parts, and thus, such constraints were simply removed. This did not produce a problem for the configuration task, as the relationships were mainly conflicts and the conflicting, lacking package was not available to be installed to the device. However, this becomes a problem for the reconfiguration task. The package list of Familiar distribution is on the Internet and it only lists the packages currently available. Packages that are no longer available are removed from the list. When reconfiguring the system, the new package to be installed may have a conflict with a package already installed on the device but no longer listed in the model. In this case, the conflict would be ignored according to our model and it would be possible to install an invalid configuration. This is an issue that needs further studies.

Feature modeling. There were some packages included in Familiar, whose names started with “task-“. These packages consisted of a set of different packages to make some basic installations easier for the end user. For example, package *task-x* includes all the packages that together implement the Linux graphical user interface called X. They should be studied further, as it seems that they provide a form of feature or function modeling.

Installation order. There were no means to model this information using PCML. When modeling the Familiar package listing with PCML, the installation order part of the information of disappeared. It can be argued that the installation process is separate from the configuration process and that the information should not be in the model in the first place. However, in the case of software, it can be claimed that these processes are commonly intertwined and the separation should not be done. In addition, modeling the information on installation order seems to be closely connected to reconfiguration, as it is essential to capture the configurations and the transitions from one configuration to another. This topic requires further work.

Configuration optimization. In section 3.2, the package relationships *recommends* and *suggests* were briefly mentioned. These can be seen as related to configuration optimization. One solution [12] would be to install the packages recommended and suggested every time a package recommending them is installed. This strategy of maximizing the configuration is not the best solution when configuring software for handheld devices. The limited size of the device does not encourage installing all recommended and suggested packages automatically without consulting the user. On the other hand, simply ignoring these relationships and thus minimizing the configuration, as was done in our approach, leaves this information totally unused. It should be studied further in which way the information should be used appropriately.

Replacement, reconfiguration. We also left out the concept of replacement from our model. The input data varied so wildly with regard to this respect and we could not reliably conclude what meaning the replaces relationship should be given. The main interpretations is, as explained in section 3.2, that the replacing package provides the functionality of the mentioned package but also overwrites it at least partially. This operation then may break existing dependencies from other packages. In addition, if the user wants to install the replaced package back on the system, again rewriting some files, she may break the whole system as the replaces relationship is only defined in one direction. In this case study, we did not research this problem further and we see this as a reconfiguration problem to be studied.

Implication and reconfiguration. Using classical implication to model the depends relationship does not seem to correspond to the way dpkg operates. When user removes a package from the system, dpkg checks that the removal does not break any existing dependencies. However, the packages installed initially just to satisfy the constraints of this package, normally due to depends- relationships, remain on the system. After a while, it is possible that the system consists of a large amount of packages, which have no justification for their existence. In a system with limited size, they easily waste resources. In order to capture this justification aspect, a new connective is needed in the constraint language that has similar properties as the rules in the weight constraint language used for formalizing configuration knowledge in [14].

Input version numbers. The version numbering of Familiar is quite versatile and due to the many different conventions in use, it is possible that in some cases a simple string comparison of version numbers may produce false results. This is, however, a problem that can only be solved by changing the version numbering conventions and cannot be simply solved by a modeling language.

6 CONCLUSIONS

We have presented a case study of modeling a configurable software product family, Linux Familiar, with a configuration modeling language designed for representing the structure of a physical product. Findings from this case study suggest that configuration modeling of a software product can be carried out to large extent using such a language. Thus it can be used as a basis for modeling and configuring software product families without a need to develop a radically new language for this purpose. However, there remain some important areas where further research is needed. In the case product, some phenomena strongly suggested that modeling them as functions or features, resources and optimality criteria, familiar from the configuration domain, would increase the understandability and usefulness of the models. In addition, there is some evidence that deeper models of versions of components and reconfiguration knowledge, not usually covered by configuration models, should be supported. In addition to researching these modeling questions, the case study should be continued by completing the model and by empirically testing its validity and the efficiency of the configurator support. In order to investigate whether the findings of this case study can be generalized, more software products from different application domains should also be investigated.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support of Technology Development Centre of Finland and Academy of Finland (grant number 51394). In addition, we thank Juha Tiihonen and Andreas Anderson for providing the configurator used in this research and for their help in using it.

REFERENCES

[1] J.Bosch, *Design and use of software architectures - adopting and evolving a product-line approsch,* Addison-Wesley, 2000.

[2] T.Männistö, T.Soininen, and R.Sulonen, ‘Modelling configurable products and software product families’, *in: IJCAI’01 Workshop on configuration,* 2001.

[3] T.Syrjänen, A rule-based formal model for software configuration. Master’s thesis.(2000). Helsinki University of Technology:

[4] D.Sabin and R.Weigel, ‘Product configuration Frameworks—A survey’, *IEEE intelligent systems & their applications,* **13**, 42-49, (1998).

[5] T.Soininen, J.Tiihonen, T.Männistö, and R.Sulonen, ‘Towards a General Ontology of Configuration’, *AI EDAM,* **12**, 357-372, (1998).

[6] F.P.Brooks, *No silver bullet -- Essence and accident in software development,* IFIP, 1986.

[7] J.Estublier, ‘Software configuration management: A roadmap’, *in: Proceedings of 22nd International Conference on Software Engineering (ICSE00), The future of software engineering,* ACM Press, 2000.

[8] T.Syrjänen, ‘Version spaces and rule-based configuration management’, *in: IJCAI’01 Workshop on configuration,* 2001.

[9] I.T.Bowman, R.C.Holt, and N.V.Brewster, ‘Linux as a case study: Its extracted software architecture’, *in: Proceedings of ICSE’99,* 1999.

[10] http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt

[11] J.Tiihonen, T. Soininen, I. Niemelä and R. Sulonen, ‘Empirical testing of a weight constraint rule based configurator’ *In Proceedings of the ECAI Workshop W02 on Configuration,* 2002

[12] T. Syrjänen: ‘Optimizing Configurations’ *In Proceedings of the ECAI Workshop W02 on Configuration,* 2000

[13] http://handhelds.org/familiar/

[14] T. Soininen, I. Niemelä, J. Tiihonen and R. Sulonen. ‘Representing Configuration Knowledge With Weight Constraint Rules’. *In Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, 2001.

[15] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, ’The Koala Component Model for Consumer Electronics Software’, IEEE Computer, 33, 78-85, 2000

[16] K. Czarnecki and U. W. Eisenecker, Generative Programming, Addison-Wesley, 2000

[17] N.Medvidovic and R.N.Taylor. ‘A classification and comparison framework for software architecture description languages’, *IEEE Transactions on software engineering,* **26**, 70-93, 2000

Configuration for mass-customisation and e-business

Ying-Lie O[1]

Abstract. Mass-customisation and e-business impose new requirements on trading. In mass-customisation, products are variations of configurations in a product family. Model-based configuration can be seen as a combination of product modelling and solving the configuration problem.

The product model can be presented using modelling constructs. Different constructs allow different kinds of selection. Instead of requiring the customer to choose from technical specifications, understandable characteristics are used.

The problem solving part is finding the configuration by a combination of selection and approximate matching. An incremental approach allows customer interaction. At each step, the customer may select components and characteristics. In the resulting matching, the list of components and non-matching characteristics are indicated. The proposed approach is attempted to be simple, such that customers can understand the proposed solutions of the specified characteristics.

1 INTRODUCTION

Nowadays, there is an increasing demand on products to customers order. These products are not fully tailored in the conventional way, but are ”variants”. This is called mass customisation, involving variations in a product family. There are three types of compositions:

- *Pick-to-order* is providing the components selected by the customer. The customer is responsible for making the product ready to use. Example: Home AV (audio-video) systems are sound systems, television sets, and combinations of these intended for non-professional use in an average size living room.
- *Assemble-to-order* is assembling of products according to customers configuration from components. Example: Office computers are computer systems for one or more persons for office work in an office or at home.
- *Make-to-order* is the production to customers specification according to existing product type definitions. Example: digital printworks are the preparation, printing, and postprocessing of a document or artwork to a printed product using professional digital printing equipment.

Mass-customisation and e-business impose new requirements on trading. Customers must be able to specify their requirements on-line without knowing details of product design and technical specifications. Mostly, the customer has to select a specific product type first, and then select the desired parts from lists. In reply, a quote with the matching product is offered. This situation is far from being customer-friendly. It is even difficult to compare a number of similar products. In a shop, products are on display along with the most important specifications.

E-business should provide additional value by employing AI. Instead of directly selecting the parts, the customer should be able to specify characteristics of the intended product. In reply, a quote with several top-most approximately matching products are offered. The products are ordered according to their “goodness of match”, indicating the non-matching characteristics. This approach requires a suitable product modelling, and a suitable problem solving method.

2 PRODUCT AND CONFIGURATION MODELS

Product modelling and configuration have been around for some time to support manufacturing. It is used for design, production, and service of the product during its life-time. Product data management and product models describe the product at certain stages “as is”. The configuration problem is finding the combination of product components such that it matches the product requirements.

2.1 Product models

There are many product modelling methods, some are specifically tailored for a certain industry or family of products. Most product models represent the hierarchical composition of the product. The bill of materials (BOM) gives the composition of an individual product. Each component is represented by a code or serial number. Associating the components of the BOM with its functional descriptions [9] gives a more detailed description. The BOM can be extended to support choices in assemble-to-order products [3] by including the list of possible components.

STEP (Standard for the Exchange of Product Model Data) is a conceptual standard for representing the configuration of individual products based on standard data. The STEP method and the specification language EXPRESS is useful for the definition of general models of products [2], including product variants [8]. The EXPRESS language can be mapped to the object-oriented UML (Unified Modeling Language) [1].

2.2 Configuration models

Configuration systems are mainly intended for supporting the manufacturing process, in particular product design. These systems are mainly focussed on problem solving [5]. There are many product configurations methods [10]. Early systems were rule-based systems, but the number of rules tend to explode, and the relation between rules and components of the product are not as clear.

In model-based systems, there is a separation between knowledge about components and the way it is used. Commonly used logic-based methods are description logics and the constraint-based satisfaction problem. A combination of these methods is a rule-based language equipped with description logics [11].

[1] University of Twente, Enschede, email: ying@cs.utwente.nl

Model-based configuration can be seen as a combination of product modelling and reasoning to solve the configuration problem. A structured modelling procedure [4] by decomposition and analysis of assembly relations allow detailed specification.

A conceptual model for the configuration of mass-customisable products employing the UML notation has been proposed by [6]. It includes consistency and validity checks and can be automatically translated to an executable logic representation for a knowledge base.

3 CONFIGURATION MODEL FOR MASS-CUSTOMISATION

Product configuration is a combination of product modelling and solving the configuration problem. In most configuration systems, customers must enter product specifications and select components from detailed lists. This requires sufficient knowledge about the product. Instead, customers should be able to specify their requirements. To get more precise, a terminology is employed. The terminology, or more general *ontology* can be extracted from existing e-business data [7] such as online catalogs. Unfortunately, most of the results regard terminology for component names, component brands, and technical specification.

The product *characteristics* is an intermediate between the customer requirements and product specification
$specification \longrightarrow characteristics \longleftarrow requirements$.
Each component has a number of characteristics. The characteristics specified by the customer are matched to that of the components. The resulting configuration is an instance of a product type.

This model is specifically intended for mass-customisation, thus limited to variations of a product family. In most product configurators, the complete set of characteristics must be specified beforehand. Here, an incremental approach at selection level is proposed. This approach allows customer interaction and update of the requirements. At each step, the customer may select components and characteristics. The proposed approach is attempted to be simple, such that customers can understand the proposed solutions of the specified characteristics.

3.1 General model of product families

Product families are modelled in a hierarchical way. Each product family consists of several clearly identifiable product types. This should limit the number of possible variations, and thus reducing the complexity. The properties of the general model are:

- A *product family* consists of mutually independent product types that may consist of other product types.
- The structure of a *product type* is a composition of subassemblies and parts. These components may be ready parts or produced according to specification.
- Matching of the user requirements to the product specification is performed through an intermediate terminology, the *product characteristics*.
- Each component has characteristics associated to its functionality and technical specification.
- The customisation is in the selection of components and characteristics. A component may be mandatory or optional.

The general model of product families as depicted in Figure 1 employs the UML notation for static structure diagrams. The modelling constructs are described below.

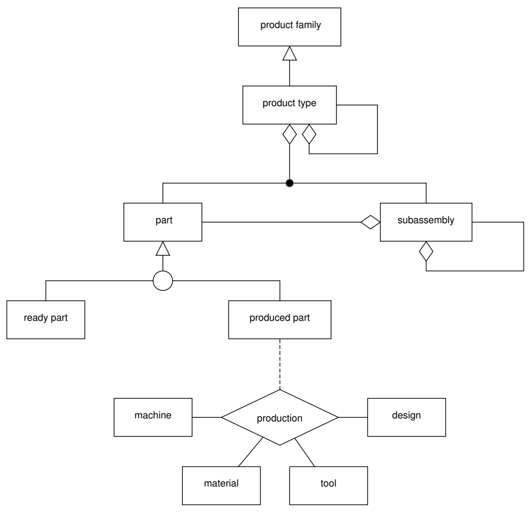


Figure 1. General model of the configuration of a product family consisting of metaclasses

- A *rectangle* represents an object *class* by its name. A class consists of objects, while a *metaclass* indicated by the $<<\ \ >>$ signs consists of classes. Metaclasses are used to represent the product and component families. Product and component models are given by classes, the products are the object instances.
- The second compartment in a divided rectangle gives the *attributes* of the class. The *product characteristics* are represented by attributes.
- The hollow triangle $\triangle$ and circle $\circ$ on the connecting line represent shared *generalisation*. Child components inherit properties from the parent component. The letter “d” (disjoint) in the circle indicates that there may be only one child in an instance. In the top layers of a product model, a *product family* is divided into *product types* using disjoint generalisation. A product type may consists of other product types.
  Selection components of a part is represented by generalisation. A *part* may be ready parts or produced parts.
- The hollow diamond $\diamond$ and bullet $\bullet$ on the connecting line define *aggregation*. An expandable *composition of components* is given by aggregation. A *product type* consists of subassemblies and parts. A *subassembly* may consist of subassemblies and parts.
- The solid diamond ⧫ and bullet $\bullet$ on the connecting line represent composition, a strong form of aggregation into a *composite class*. In product models a *composite component* is built of subcomponents that solely belong to that component and the composition yields its functionality as a whole. This specific property may be valid for some subassemblies, and is therefore not indicated as metaclass in the general model.
- A large diamond connected with lines to rectangles represent an *association*. The *association class* is given by a rectangle connected with a dashed line. The production of a *produced part* is represented by an association of components required in the production.
- The permitted number of components is specified by the multiplicity as a range of natural numbers. Thus, $1..n$ means 1 to n, $1..1$ means exactly 1, $0..1$ means at most 1.

The general product model in Figure 1 consists of metaclasses only, therefore the $<<\ \ >>$ signs are left out. Specific composition properties that apply on classes are therefore not visible.

3.2 Product characteristics

As an attribute, a characteristic has an identifying generally meaningful name and a domain of possible values as a user-friendly translation of the technical specification.

Characteristics are preferably objective, and the domain consists of comparative terms. This makes the characteristics invariant to rapid changes, for instance in technology. $quality\{plain, good, high\}$ in printing gives the resolution of the printer and grain size of the paper; $size\{small, medium, large\}$ related to a PC disk gives the storage size with respect to current technology; $speed\{medium, fast, super\}$ specifies the PC computing power, and is related to the technical specification of the CPU frequency and internal memory.

Inheritance and association of the characteristics are determined by the modelling constructs. *Child components* in a *generalisation* inherit properties from the parent component and may have additional specialisation characteristics
$\{child\,chars\} = \{parent\,chars\} \cup \{specialised\,chars\}$.
A *composite component* has all the characteristics of its subcomponents $\{composite\,chars\} = \{component_1\,chars\} \cup \cdots \cup \{component_n\,chars\}$.
Similar to the composite component, a *composition of components* has all the characteristics of its selected components. Also similar to the composite component, the *produced part* has all the characteristics of its associated components.

Except for inheritance properties, the scope of the characteristics may be *global*, or *local*. In the global mode, all characteristics of the same name get the same value
$char_x = a \Rightarrow \forall$ components $component\,char_x = a$.
In the local mode, it is limited to the selected component. In a generalisation, characteristics regarding the specialisation are always local. The mode may be changed during the selection process. For instance, the desired colour of a product may be set global at the beginning, and set to local regarding a specific part.

3.3 Selection and matching

The problem solving part is finding the configuration of the desired product by a combination of *selection* and *approximate matching*.

The approximate matching problem is formulated as finding the components with the most matching characteristics to the user requirements. This will result in a list of products, that may have several non-matching characteristics. Customisation by selection occurs at three levels:

1. Class-level in *aggregation* constructs by selection of a combination of desired components.
2. Class-level in *generalisation* constructs by selection from a list of possible components.
3. *Object*-level in all classes by selection of an object instance in each class.

Matching the user requirements to the product characteristics are set-based operations. Set union $\cup$ gives all the characteristics, and set intersection $\cap$ gives the matching characteristics.

The incremental approach allows (de)selection of both components and characteristics at each step. Non-matching characteristics are indicated, allowing the customer to modify the selections.

The selection of components yields the associated characteristics and their value range.
$\forall$ selected components $\{total\,chars\} = \cup\,\{total\,chars\} \cup \{component_1\,chars\} \cup \cdots \cup \{component_n\,chars\}$.
On the other hand, the selection of characteristics gives the matching components and non-matching characteristics.
$\forall$ selected characteristics $\{total\,chars\} = \cup\,\{total\,chars\} \cup \{char_1\} \cup \cdots \cup \{char_n\}$.
Given the $\{total\,chars\}$ $\exists$ matching components with $\{component\,chars\} \cap \{total\,chars\} \neq \emptyset$. Non-matching characteristics are not present in the set of selected characteristics $\{nonmatch\,chars\} : \{component\,chars\} \notin \{total\,chars\}$.
This process is alternately repeated on components that have a path to the selected product type.

The selection starts with choosing from an index of product families and product types. As in a large department store, customers are usually able to find the department of a particular product family.

1. Class-level top-down matching to determine the product types. The requirements of the product types generally regard the intended use of the product.
2. Class-level characteristics from the selected product types are retrieved and presented for selection. Product types may be deselected.
3. Then for each selected product type, advance downwards on class-level. This involves tracing the connected paths to the product types. The selection procedure is carried out for aggregation constructs and generalisation constructs.
4. Class-level bottom-up analysis of association classes and composition classes that are connected to the selected classes.
5. Class-level characteristics from the selected components are retrieved and presented for selection. At this stage, characteristics may be altered locally only for a particular component.
6. Object-level selection from the list of matching objects of all remaining classes.
7. Determine a list of matching product compositions.

In the above, in the first three steps, the characteristics are global, and may become local in the last four steps.

3.4 Reconfiguration

Reconfiguration is altering an existing product configuration for replacement, upgrading, or changing the functionality. There are two modes of reconfiguration:

Addition or replacement of components that does not change the remaining of the existing configuration such as addition of a component in an aggregated class, replacement of a child component in a generalisation class, or replacement of an object instance. Most of the functionality and global characteristics remain the same.

Changes that affect the configuration such as upgrades of an earlier version of a product type typically regard composition classes and aggregation classes. This can be considered as a change of the product type. Therefore, an "upgrade model" containing the possible replacements is required.

4 EXAMPLES

Each of the composition type is illustrated by a familiar example. To avoid cluttered diagrams, the models are only partly presented and detailed. For the same reason, the characteristics that mostly yield the "customisation" are indicated by empty compartments. Also, the multiplicities are left out.

To illustrate the matching process, the most relevant characteristics are presented in tables. In the first table, the characteristics are described. In the next table, the selection process is illustrated by showing the requirements characteristics entered by the customer, the matching product type and associated characteristics.

4.1 Home AV systems

"Home AV systems" is an example of pick-to-order products. It is

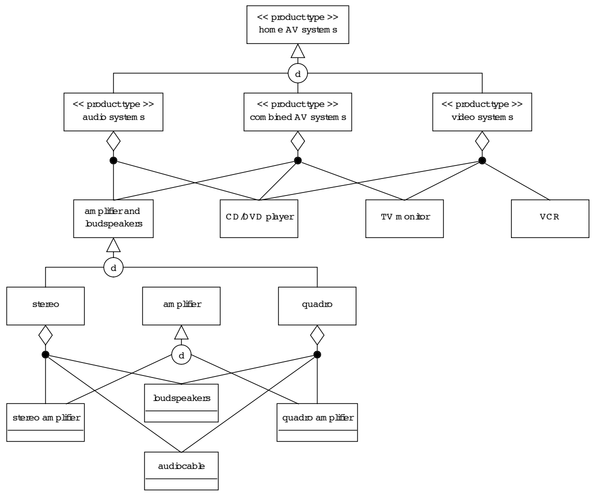


Figure 2. Part of the configuration model of home AV systems

a product type of the product family "AV systems". Usually, components have matching connectors. In Figure 2 part of this model is presented, along with the most relevant part of the characteristics in Table 1.

Table 1. Part of the characteristics of home AV systems

class	construct multiplicity	characteristics constraints
home AV systems	product type	use = non-professional
audio systems	product type	AV = audio
AV systems	product type	AV = (audio, video)
audio systems	aggregate	AV = audio
CD/DVD player	part	type {CD, DVD}
television	part	AV = video
amplifier and loudspeakers	generalisation 1..1	type {stereo, quadro} power {low, med, high}
stereo	aggregate 0..1	type = stereo
quadro	aggregate 0..1	type = quadro
amplifier	generalisation	quality {plain, good, high} power {low, med, high}
loudspeaker	part 2..2	quality {plain, good} power {low, med} connector = RL pin
stereo amplifier	part 1..1	quality {plain, good, high} power {low, med, high} connector = RL pin
audiocable	part 2..2	connector {RL pin, coax} connector1, connector2

In pick-to-order, there usually is an abundance of choices on object-level. Therefore, additional characteristics such as brand, and colour may help to restrict the choices.

1. The selection starts at the level of home AV systems.
2. The choice of AV = audio yields 2 choices, one with a non-matching characteristic.
3. Advancing downwards is rather straight-forward, allowing the selection in "amplifier and loudspeakers" and "CD/DVD players".
4. There are no association classes and composition classes.
5. Selection of characteristics, for instance a local value is selected: loudspeaker.quality = good.
6. Selection of stereo amplifiers, loudspeakers, and CD players with the selected characteristics.
7. The matching products are stereo amplifiers, "loudspeakers (2)", "audio cables (2)", and "CD players" with matching connectors.

Table 2. Example of the selection and matching of home AV systems

requirements characteristics	class	characteristics of components
AV = audio deselect	audio systems AV systems AV systems	AV = audio AV = audio

type = CD
type = stereo
 	
type = stereo
	CD/DVD player
	amplifier and loudspeakers
	stereo
		type = CD
		type = stereo
		power {low, med, high}
		type = stereo

quality = good
power = med
–
	loudspeaker	quality = good
		power = med
		connector = RL pin

quality = high
–
–
	stereo amplifier	quality = high
		power = med
		connector = RL pin

–
–
	audiocable	connector1 = RL pin
		connector2 = RL pin

4.2 Office computers

A familiar example of assemble-to-order products are "office computers" a product type of the product family "Computers". This prod-

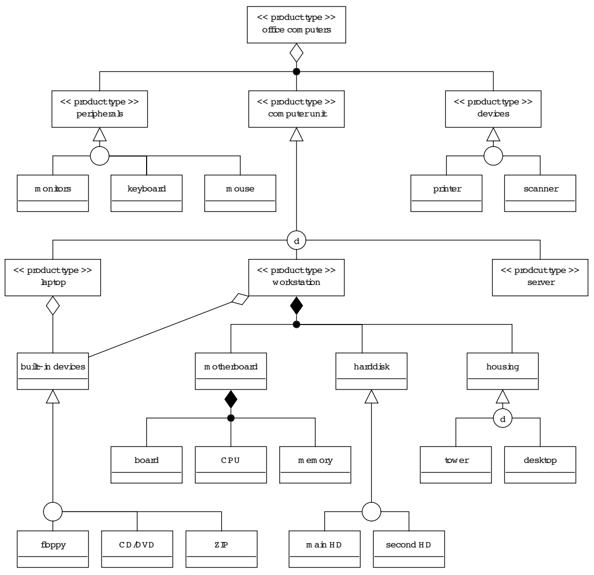


Figure 3. Part of the configuration model of office computers

uct type is further divided into specific product types. In addition to the assemble-to-order part, there are pick-to-order parts to combine the configured system with other devices. It is partly represented in Figure 3, with the most relevant characteristics in Table 3.

Table 3. Part of the characteristics of office computers

class	construct	characteristics
	multiplicity	constraints
office computers	product type	use = office work
computer unit	product type	main unit
peripherals	product type	addition
devices	product type	addition
laptop	product type	type = client
workstation	product type	place = fixed, type = client
server	product type	place = fixed, type = server
workstation	composition	type
built-in devices	generalisation	type {flop, CD/DVD, ZIP}
	0..*	slot $\leq$ housing.slot
motherboard	composition	type
	1..1	
board	part 1..1	
CPU	part 1..1	speed {med, fast, super}
memory	part 1..4	speed {med, fast, super}
harddisk	generalisation	main HD = 1..1
	1..2	second HD = 0..1
main HD	child 1..1	size {small, med, large}
		speed {med, fast}
second HD	child 0..1	size {small, med}
		speed {med, fast}
housing	generalisation	type {tower, desktop}
	1..1	slot 1..*
tower	child	second HD = 0..1, slot = 5
desktop	child	second HD = 0, slot = 2

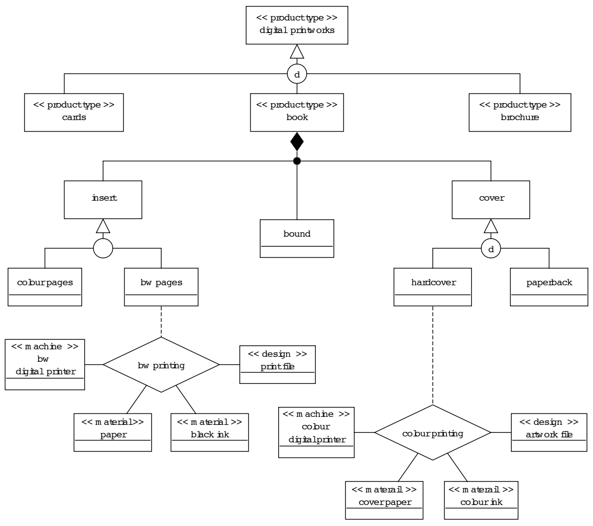
Table 4. Example of the selection and matching of office computers

requirements	class	characteristics
characteristics		of components
main unit	computer unit	main unit
place = fixed	workstation	place = fixed
type = client		type = client
–	built-in devices	slot = 2
type = CD/DVD		type = CD/DVD
type = ZIP		type = ZIP
–	workstation	type
–	motherboard	type
–	board	
speed = fast	CPU	speed = fast
speed = fast	memory	speed = fast
second HD	harddisk	second HD = 1
size = large	main HD	size = large
speed = fast		speed = fast
size = med	second HD	size = med
speed = med		speed = med
–	housing	type = tower
–	tower	second HD = 1, slot = 5

In this example, only the assemble-to-order part is tracked.

1. The selection starts with “computer unit”.
2. The choice of place = fixed, type = client yields the only product type “workstation”.
3. Advancing downwards only regards the “built-in devices”. The choices of “floppy”, “CD/DVD”, and “ZIP” require 2 slots.
4. Composition classes that are connected to “workstation” are “motherboard”, “harddisk”, and “housing”.
5. In the selection of characteristics, the “second HD” has less characteristics than the “main HD”.
6. Object-level selection from all remaining classes. There may be additional constraints, such as the number of slots for “built-in devices” and “hard disks” that fit in the “housing”.
7. Find matching “workstations” with fast performance, 2 “harddisks”, “tower housing”, and the desired “built-in devices”.

4.3 Digital printworks



<< product type >> digital printworks
d
<< product type >> cards
<< product type >> book
<< product type >> brochure
insert
bound
cover
colour pages
bw pages
d
hardcover
paperback
<< machine >> bw digital printer
bw printing
<< design >> printfile
<< material>> paper
<< material >> black ink
<< machine >> colour digitalprinter
colourprinting
<< design >> artwork file
<< material >> coverpaper
<< material >> colour ink

Figure 4. Part of the configuration model of digital printworks

Table 5. Part of the characteristics of digital printworks

class	construct	characteristics
	multiplicity	constraints
digital	product type	printing = digital
printworks		ink = cartridge
book	product type	
card	product type	
brochure	product type	
book	composition	quality {plain, good, high}
insert	generalisation	colour {colour, bw}
	1..2	quality {plain, good, high}
cover	generalisation	type {hard, paperback}
	1..1	quality {plain, good, high}
bound	part 1..1	if cover.type = paperback
		then bound = adhesive
		if cover.type = hard
		then bound = {stitch, sew}
bw pages	association 1..*	colour = bw
printfile	design	quality = good
		paper size {A3, A4, A5}
paper	material	quality {plain, good, high}
		paper size {A3, A4, A5}
black ink	material	
digital printer	machine	colour = bw
cover	association 1..1	colour = colour
artworks	design	quality = good
		paper size {A3, A4, A5}
cover paper	material	quality {plain, good, high}
colour ink	material	colour {3-colour, 6-colour}
digital printer	machine	colour = colour

"Digital printworks" is an example of make-to-order products. It is a product type of the product family "Press works". Printed pages are generated by printing machines from the file delivered by the user. As partly presented in Figure 4, the product type is further divided into different types. The most relevant characteristics associated to the figure are given in Table 5.

Table 6. Example of the selection and matching of digital printworks

requirements characteristics	class	characteristics of components
book	book	
quality = good	book	quality = good
colour = bw	insert	colour = bw
–		quality = good
type = hard	cover	type = hard
–		quality = good
–	bound	bound = $\{$stitch, sew$\}$
–	bw pages	colour = bw
–	printfile	quality = good
paper size = A5		paper size = A5
quality = good	paper	quality = good
–		paper size = A5
black	black ink	
–	digital printer	colour = bw
–	hard	colour = colour
–	artworks	quality = good
–		paper size = A5
quality = high	cover paper	quality = high
colour= 3-colour	colour ink	colour= 3-colour
–	digital printer	colour = colour
bound = sew	bound	bound = sew

The product type "book" is taken as an example to illustrate the selection and matching in a composite component and produced parts.

1. The selection of the product type "digital printworks" has been made.
2. The product type "book" is selected.
3. Going downwards only regards this product type.
4. The components in the composite component "book" are related by the way it is produced. The associated components in the "bw pages" and "cover" represent material and settings of the printing process.
5. Selection of the characteristics of the "bound" is limited by the choice of the "cover".
6. Object-level selection basically is the selection of the "paper".
7. Matching products mainly regard the choices in the printing process and the use of paper and ink. There may be some matching between the printing machines and paper type and size.

5 CONCLUSION

A simple incremental product configuration method for mass-customisation has been proposed. The product model consists of modelling constructs for each configuration feature. For customer requirements, characteristics are used instead of technical specification. The characteristics selected by the customer are then used to find matching components that have a connection path to the selected product type. The approximate matching is performed using simple set-based methods. The method allows non-matching characteristics.

From the examples it is found that in some cases business rules are still needed. It should be further investigated how such rules can be replaced by a set construct.

A possible improvement of the matching process, is by ranking the characteristics in a sequence of importance as a partial order of sets $\{charset_1\} \succeq \ldots \succeq \{charset_n\}$.
The selection and matching then becomes a combination of tracking the paths of selected and associated components, ordering of the characteristics, and matching these ordered characteristics.

Further research should also include consistency analysis of the proposed solutions. The method should then be improved to handle conflicting situations. It should also be investigated, whether additional constraints or rules are needed to maintain consistency in conflicting situations.

Another important aspect in the application of this method is model management. It is expected that if the modelling constructs are applied consistently, then the model definition part will not pose any problems. However, problems are expected in the definition of the characteristics. Even using a proper ontology, maintenance is still elaborate. Updates of ontologies, and merging different ontologies is still a research topic at this very moment.

It is also of interest to include different ways of representing the modelling constructs for specification and implementation. Also, the possibilities to extend this simple model to a full product design model should be considered.

ACKNOWLEDGEMENTS

The author is indebted to Rob van de Weg for his valuable detailed comments, and colleagues of the group of Information Systems for their inspiring discussions.

The UML figures were drawn using TCM (Toolkit for Conceptual Modelling) partly developed by this group, available at $www.cs.utwente.nl/\ tcm$ under the GNU General Public License.

REFERENCES

[1] F. Arnold and G. Podehl, 'Best of both worlds - A mapping from EXPRESS-G to UML', in *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Selected Papers*, eds., J. Bézivin and P.-A. Muller, volume 1618 of *LNCS*, pp. 49–63. Springer Verlag, (1999).

[2] M. Ashworth, M.S. Bloor, A. McKay, and J. Owen, 'Adopting STEP for in-service configuration control', *Computers in Industry*, **31**, 235–253, (1996).

[3] J.W.M. Bertrand, M. Zuijderwijk, and H.M.H. Hegge, 'Using hierarchical pseudo bills of material for customer order acceptance and optimal material replenishment in assemble to order manufacturing of non-modular products', *Int. J. Production Economics*, **66**, 171–184, (2000).

[4] P.Y. Chao and T.T. Chen, 'Analysis of assembly through product configuration', *Computers in Industry*, **44**, 189–203, (2001).

[5] B. Faltings and E.C. Freuder, 'Special issue on configuration', *IEEE Intelligent Systems*, **13**(4), 29–85, (1998).

[6] A. Felfernig, G.E. Friedrich, and D. Jannach, 'Conceptual modeling for configuration of mass-customizable products', *Artificial Intelligence in Engineering*, **15**, 165–176, (2001).

[7] A. Kayed and R.M. Colomb, 'Extracting ontological concepts for tendering conceptual structures', *Data & Knowledge Engineering*, **40**, 71–89, (2002).

[8] T. Männistö, H. Peltonen, A. Martio, and R. Sulonen, 'Modelling generic product structures in STEP', *Computer-Aided Design*, **30**(14), 1111–1118, (1998).

[9] T. Männistö, H. Peltonen, T. Soininen, and R. Sulonen, 'Multiple abstraction levels in modelling product structures', *Data & Knowledge Engineering*, **36**, 55–78, (2001).

[10] D. Sabin and R. Weigel, 'Product configuration frameworks – a survey', *IEEE Intelligent Systems*, **13**(4), 42–49, (1998).

[11] T. Soininen and I. Niemelä, 'Developing a declarative rule language for applications in product configuration', in *PADL'99 Practical Aspects of Declarative Languages Proc. First International Workshop*, ed., G. Gupta, volume 1551 of *LNCS*, pp. 305–319. Springer, (1998).

Empirical Testing of a Weight Constraint Rule Based Configurator

Juha Tiihonen[1], **Timo Soininen**[1], **Ilkka Niemelä**[2], and **Reijo Sulonen**[1]

Abstract. In this paper we first describe a configurator implementation based on a practically important subset of a synthesized ontology of configuration knowledge. The underlying configuration modeling language has been provided with a declarative semantics by mapping it to weight constraint rules, a form of logic programs. Three issues important for efficiency of the implementation are addressed: off-line compilation of configuration models, limiting a configuration to a finite size in a semantically justified way, and breaking symmetries in the set of configurations. The second part of the paper takes a step in the direction of thorough empirical testing of configurators. We define a relatively modeling-language-independent method for testing configurators based on the idea of simulating a naïve user inputting random requirements to a configurator. We test the configurator empirically on batch-mode sales configuration of four real products with progressively larger and thus more restricting sets of random user requirements. The results indicate that our configurator is efficient enough for practical use.

1 INTRODUCTION

Several formal models of configuration knowledge and tasks based on, e.g., constraint satisfaction problems (CSP) and different logical formalisms have been proposed, e.g., [1,2,3,4] and implemented, e.g., [4,5,6,7,8]. For several of these, the configuration problem has been shown to be at least NP-hard [2,4,9,10]. In other words, the configuration task requires in the worst case at least an exponential amount of time in the size of the problem. However, conventional wisdom in the configuration community is that solving configuration problems is relatively easy and does not exhibit this kind of behavior. There are some documented results on the efficiency of configurators [4,6,7,8], but systematic and wide range empirical testing of configurators on real products that would show whether the wisdom is, indeed, wisdom, is still lacking.

In this paper we take a step in the direction of thorough empirical testing of configurators. We briefly describe a configurator implementation, define a general test methodology for configurators, and provide results on the efficiency of our implementation.

Our configurator uses a modeling language based on a practically important subset of a synthesized ontology of configuration knowledge [11]. The language has a declarative semantics provided by mapping it to weight constraint rules, a form of logic programs [2,12]. The configurator uses this mapping to translate the modeling language to weight constraint rules. It then uses a state-of-the-art implementation of weight constraint rules, Smodels [12], for computing configurations satisfying user requirements. Three issues important for efficiency of the configurator are addressed: off-line compilation of configuration models, limiting a configuration to a finite size in a semantically justified way, and symmetry breaking.

We have modeled four real products from a sales configuration point of view. The case products are characterized and the configurator is empirically tested on batch-mode configuration of these products. We define a relatively modeling-language-independent method for testing configurators based on the idea of simulating a naïve user inputting random requirements to a configurator. This is accomplished by randomly generating progressively larger and thus more restricting sets of user requirements that are not locally conflicting. Results are given on the number of correct configurations found and the time it takes to find the first and all configurations satisfying a set of random requirements, or to show that no such configuration exists.

The rest of the paper is structured as follows: In section 2 the modeling language and its semantics are outlined and in section 3 the configurator implementation based on the language is described. In section 4 the testing method and the case products are described and the test results are provided. Finally, in section 5 we discuss and compare our implementation and results with related work and in section 6 we present conclusions and topics for further work.

2 MODELING LANGUAGE

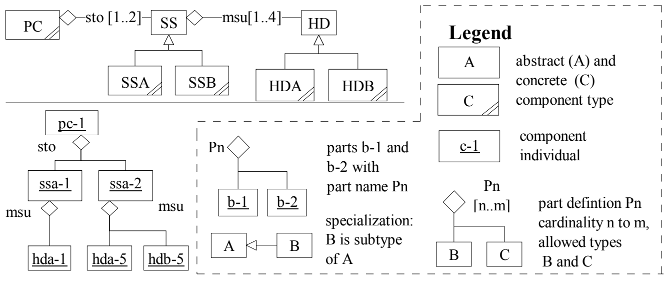
In this section we briefly describe *PCML*, the product configuration modeling language of our configurator, and outline its semantics. For more information on the modeling language and its implementation, see http://www.soberit.hut.fi/pdmg/empirical_cfg/ and [2].

The main concepts of PCML are *component types*, their *compositional structure*, *properties* of components, and *constraints*. Component types define intensionally the characteristics (such as parts) of their *individuals* that can appear in a configuration. A component type is either *abstract* or *concrete*. Only an individual directly of a concrete type is specific enough to be used in an unambiguous configuration. A component type defines its direct parts through a set *of part definitions*. A part definition specifies a *part name*, a non-empty set of *possible part types* (*allowed types* for brevity) and a *cardinality*. A component type may define properties that parameterize or otherwise characterize the type. A *property definition* consists of a *property name*, a *property value type* and *a necessity definition.* Component types are organized in a *taxonomy* or class hierarchy where a *subtype* inherits the property and part definitions of its *supertypes* in the usual manner.

Figure 1 illustrates the concepts through an example. A server PC has 1 to 2 storage subsystems. There are two kinds of storage subsystems, SSA and SSB. A storage subsystem has 1 to 4 hard disks. The hard disk types in use are HDA and HDB. This is modeled as follows (the upper part of Figure 1): Concrete component type `PC` has a part definition with part name `sto`, cardinality 1..2 and allowed type `SS`. Component type `SS` is abstract and has two concrete subtypes `SSA` and `SSB`. `SS` has a part definition `msu` (mass storage units) with cardinality 1 to 4 and allowed type `HD`. Type `HD`

[1&2] Helsinki University of Technology, Dept of Computer Science and Eng.,
[1] Software Business and Engineering Institute, P.O.B. 9600, FI-02015 HUT
[2] Lab. for Theoretical Computer Science, P.O.Box 5400, FI-02015 HUT
[1&2] {Juha.Tiihonen, Timo.Soininen, Ilkka.Niemela, Reijo.Sulonen}@hut.fi

is abstract and has two concrete subtypes `HDA` and `HDB`. The lower part of Figure 1 shows a configuration where individual `pc-1` of type `PC` has as a part with part name `sto` two storage subsystems of type `SSA` (`ssa-1` and `ssa-2`). The individual `ssa-1` has as a part with part name `msu` one hard disk of type `HDA` (`hda-1`), while `ssa-2` has as a part with part name `msu` one hard disk of type `HDA` (`hda-5`) and one hard disk of type `HDB` (`hdb-5`).



PC
sto [1..2]
SS
msu[1..4]
HD
SSA
SSB
HDA
HDB
pc-1
sto
ssa-1
ssa-2
msu
msu
hda-1
hda-5
hdb-5
Legend
A
C
abstract (A) and concrete (C) component type
c-1
component individual
Pn
parts b-1 and b-2 with part name Pn
b-1
b-2
specialization: B is subtype of A
A
B
Pn [n..m]
part defintion Pn cardinality n to m, allowed types B and C
B
C

Figure 1. Example product

Constraints associated with component types define conditions that a correct configuration must satisfy. The first level building blocks of the constraint language are *references* to access parts and properties of components, and constants such as integers. References can be used in succession, e.g. to access a property of a part. *Tests* returning Boolean values are constructed using references, constants, and arithmetic expressions. Tests include predicates for checking if a particular referenced individual exists or is of a given type. Property references can be used with constants in arithmetic expressions that can be compared with the usual relational operators to create a test. Test can be further combined into arbitrarily complex *Boolean expressions* using the standard Boolean connectives.

The semantics of the modeling language is provided by mapping it to weight constraint rules [2]. The basic idea is to treat the sentences of the modeling language as short hand notations for a set of sentences in the weight constraint rule language (*WCRL*) [11]. A *configuration model* thus corresponds to a set of weight constraint rules consisting of ontological definitions defining the semantics of the modeling concepts and a set rules representing the configuration model. A *configuration* with respect to a configuration model is defined as a subset of the Herbrand base of the configuration model. A *requirement* is for simplicity defined as an atomic fact. A *correct configuration* is a stable model of the set of rules representing the configuration model and a *suitable configuration* is a correct configuration that also satisfies the set of requirements.

3 IMPLEMENTATION

This section first describes relevant parts of our prototype configurator and off-line compilation of configuration models. Then we discuss individual generation that limits a configuration to a finite size in a semantically justified way. Finally we discuss symmetry breaking that is important for the performance of the configurator.

3.1 Overview and implementation scope

The configurator architecture is outlined in Figure 2. The configurator is implemented in Java programming language except components smodels and lparse of the Smodels system, described below.

The configurator compiles a PCML program to a general WCRL program with variables and further to simple basic rules (*BCRL*) that contain no variables. This potentially costly two-phase compilation process is performed off-line. In the compilation, *PCML core* in the *Model manager* loads a PCML configuration model and checks it for consistency. This includes parsing the PCML file, type checking expressions and checking the configuration model for validity with respect to the language specification. The *Smodels interface* in Model manager translates the configuration model to a WCRL program. Data structures representing the PCML configuration model are saved for later use. The generated WCRL program includes sentences for ontological definitions, component type hierarchy, compositional structure, properties, and constraints as described in [2]. In addition, rules for component individuals and symmetry breaking are included, described in Sections 3.2 and 3.3. Finally, the WCRL program is translated to BCRL.

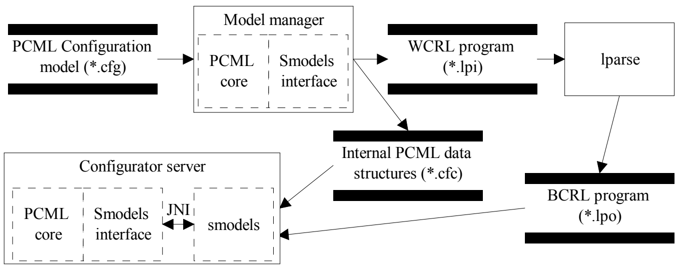


Figure 2. Configurator architecture

The configurator uses as the configuration engine an implementation of the weight constraint rule language called *Smodels*[12]. The main functionality of the Smodels system is to compute a desired number of stable models for a WCRL program. The system allows a user to give further requirements (through so-called *compute statements*) to constrain the stable models to be computed.

The Smodels system is based on a two-level architecture where in the first phase a front-end, *lparse*, compiles a WCRL program into a BCRL program. Lparse exploits efficient database techniques but does not resort to search. The search for models of BCRL programs is handled using a special purpose search procedure, *smodels*, taking advantage of special features of BCRL. The search procedure works in linear space and employs efficient search space pruning techniques and a powerful dynamic application-independent search heuristic. Smodels is implemented in C++ and offers APIs through which it can be directly integrated into other software. Smodels is publicly available at http://www.tcs.hut.fi/Software/smodels/.

The BCRL form of the configuration model is used to repetitively configure a product. The configurator server loads the data structures representing the PCML configuration model into PCML core and the BCRL program into smodels. A compute statement representing requirements is set through smodels API.

3.2 Individual generation

In this work, we take the approach that the set of individuals out of which the configuration can be constructed is pre-defined. This limits the configuration to a finite size. We decide in advance in a semantically justified way the number of individuals of each concrete type available for use in a configuration. The available individuals are represented as a set of facts stating for each individual which type it is an individual of. These facts are added to the WCRL rules representing the configuration model.

There is exactly one component type that can serve as the root of the compositional structure, referred to as the configuration type. An individual of this type, the configuration individual, serves as the root of the compositional structure. Because a maximum cardinality is defined for every part definition, we can calculate an upper bound of the number of needed individuals. First the configuration individual is generated. For each part definition of the configuration type, the number of individuals defined by maximum cardinality of the part definition is generated for each allowed concrete part type. This is performed recursively to generate part individuals for all part definitions of the types of the newly generated individuals.

The number of needed individuals can grow exponentially. For example, increasing the number of levels in the part hierarchy leads to exponential growth in the number of generated individuals when maximum cardinality at each level is at least 2. The implementation does not try to optimize the number of individuals on the basis of constraints or mutually exclusive branches of the part structure.

3.3 Symmetry breaking

Individuals of a concrete type are equivalent except for their names. *Equivalent configurations,* i.e. configurations identical except for naming, can be created by selecting different individual(s) of a concrete type as a part. This freedom of selection creates unwanted symmetries. Next, we describe two forms of symmetries and present a method used in our configurator that allocates individuals to specific part names of specific individuals in a way that breaks these symmetries.

The first form of symmetry arises when several individuals directly of a type are possible parts with a part name for an individual. For example, in Figure 3(a) type `A` has part definition `P` with cardinality 1 to 2 with type `B` as the only allowed type. There are two individuals `b-1` and `b-2` of type `B` that can be as a part with part name `P` in `a-1`. The configurations in Figures 3(b) and 3(c) are equivalent. In general, individuals can be picked in a combinatorial number of ways creating a potentially huge number of symmetries. The idea of symmetry breaking is that the possible part individuals directly of the same type are always used in a fixed order. The individuals are ordered by giving them priority rankings. A lower priority individual is not allowed in the configuration if all the higher priority individuals are not in the configuration. Symmetry breaking would thus allow only the configuration in Figure 3(b).

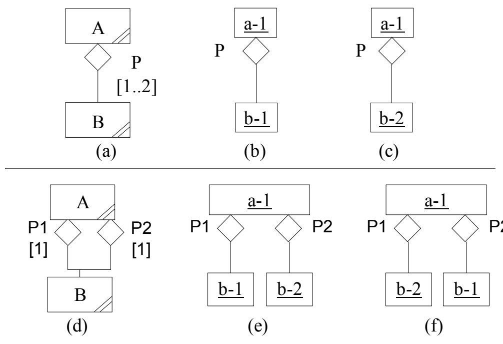


Figure 3. Symmetry breaking

The second form of symmetry is shown in Figures 3(d) to 3(f). Without symmetry breaking any individual of type B could be as a part with any part name in any individual whose type has a part definition that has B as an allowed type. For example, individuals `b-1` and `b-2` of type `B` could both be as a part with part name `P1` or `P2` in individual `a-1`. Our solution for breaking this form of symmetry is to allocate each individual to a specific individual and part name. After allocation either (e) or (f) is allowed, but not both.

4 EMPIRICAL TESTING

In this section we discuss empirical testing of configurators in general and our configurator implementation in particular. We first describe a general test methodology for configurators and continue by describing and characterizing the products used as test cases. We then specify our test setup. Finally, we provide results on the efficiency of our configurator.

4.1 Testing method

In principle, one could test a configurator by using real configuration models or by using randomly generated configuration models. Random configuration model generation could be synthetic or use real products as a seed. Another dimension is the selection between fixed and randomly generated requirements. We chose for this work real configuration models with random requirements.

There is a risk that random models without a large set of real products as a seed do not reflect the structured and modular nature of products designed by engineers. In addition, it is hard to attain a level of difficulty representative of real problems. Knowledge acquisition and modeling for a sufficient seed of real models for random model generation in a justified way would be a major task.

Random requirement generation with progressively larger and thus more restrictive sets of requirements allows one to investigate how well the configurator performs with varying sizes of requirement sets. A dramatic increase in time to find a configuration with some requirement size indicates that the problem becomes critically constrained at that point. The existence of hard configuration problems would then be revealed.

For generating random sets of requirements, we consider how the configuration model appears to the user configuring a product. There are menus (possibly multi-choice), radio buttons or check boxes to select between different alternatives. Guided with these, it is probable that the user will not break the "local" rules of the configuration model, e.g. by requiring alternatives that do not exist or by selecting a wrong number of alternatives. However, a naïve user can easily break the rules of the configuration model that refer to the dependencies of several selections.

We follow this idea by considering the configuration model as consisting of a set of “local” requirement groups. A *requirement group* (*group* for brevity) represents a set of potential requirements that a user could state. For example, a group could represent the selection of a value for the power property of an engine, or the selection of the cooling system in a compressor out of the allowed component types. Each group has a number of *requirement items* each representing a potential requirement. The number of requirements that can be generated from a group is defined by *minimum* and *maximum cardinality*. Note that cardinality applies only if the group is selected to generate requirements.

In our tests, a requirement group is created for each property and part definition of the type of each individual. For each property definition a group with maximum and minimum cardinality of one is created. A value in the domain of the property corresponds to one requirement item. If the property is optional, a requirement item that denies a value for the property is included. A part definition corresponds to one group with maximum cardinality of the part definition. Minimum cardinality is the maximum of one and the minimum cardinality of the part definition. Each potential part individual corresponds to one requirement item. If the cardinality of the part definition includes 0, a requirement item that denies all part individuals is included.

A *test case* contains a number of requirement items related to a configuration model. When generating a test case, a group is randomly selected to generate the number of requirements specified by the minimum cardinality. A group can be selected again to generate one new requirement. Group selection is repeated until the desired number of requirements has been generated. A group cannot be selected to generate requirements, if the desired number of requirements or maximum cardinality would be exceeded, or if all requirement items are already in the generated requirements.

A requirement is generated from a group representing a property by choosing randomly one requirement item. Generating a requirement for a part is slightly more complex. In our implementation, the order in which the individuals of a given type may be chosen as requirements is important due to the symmetry breaking. Therefore, a requirement is generated by randomly selecting the direct type of the part or the requirement item that denies all part individuals. If a type is chosen, the highest priority individual of that type that has not been required yet is set as the requirement. For example, `hda-1` of Figure 1 would be required before `hda-2`, as they are allocated to the same part name (`msu`) of a component individual (`ssa-1`).

4.2 Case products

We have modeled four real product families using PCML. Three products are screw compressors manufactured by Gardner Denver Oy. Each configuration model represents a complete sales configuration view of a compressor family. The models are detailed to production quality, except for some constant values. The fourth product is a 4-wheel vehicle anonymized by renaming. It was modeled for demonstration purposes and represents about half of the sales view of the product. Numerous optional parts and some constraints were excluded. Despite inaccuracies the model reflects quite well the nature of sales configuration of this configurable product.

The configuration models are characterized in Table 1. Row "Comp. types" gives the number of concrete, abstract and all component types. "Properties" specifies the total number of properties and the number of component types that specify at least one property. "Domain size" indicates minimum, maximum and average domain size of the defined properties. It also gives the number of properties with "small" domain size of 2 or 3, as the average domain sizes are strongly affected by the few large domain properties. "Part defs" specifies the total number of part definitions, the number of component types with part definitions, and the average number of allowed concrete component types. "Cardinality" specifies the number of part definitions with different cardinalities: 0 to 1, exactly 1, and others. "Constraints" specifies the number of constraints and the average number of parts or properties referenced by a constraint. In every compressor model all constraints except one had 2 or 3 references. The exceptional constraints had 262 to 347 references to enumerate allowed combinations of four to five properties. These huge constraints dominate the averages. In all configuration models, the configuration type defined all the constraints and most properties.

Compressor configuration models use almost solely properties. The most complex of these, ESVS, has 3 part definitions. Optional components without properties were modeled as properties.

Configuration model	ESVS	FS	FX	Vehicle
Comp. types c / a / tot	7 / 2 / 9	3 / 0 / 3	1 / 0 / 1	25 / 4 / 29
Properties tot / ct	24 / 5	22 / 3	18 / 1	8 /3
Domain size min – max avg / 2-3	2 – 61 5.9 / 17	2 – 51 5.5 / 14	2 – 44 5.7 / 10	2 - 22 5.8 / 5
Part defs tot / ct / allow	3 / 2 / 2	1 / 1 / 2	0 / 0 / -	16 / 3 / 1½
Card. 0-1 / 1 / max >1	0 / 3 / 0	0 / 1 / 0	0 / 0	12 / 4 / 0
Constraints tot / refs	17 / 20	14 / 25	21 / 12	7 / 2.0

Table 1. Properties of the configuration models

4.3 Test setup

Test setup is illustrated in Figure 4. A WCRL program was generated off-line for each PCML configuration model. For each test case, a new process was created to execute a batch file that executed lparse (version 1.0.4) to generate a BCRL program with a compute statement with the requirements of the test case. The output of lparse was piped to smodels version 2.26 with modifications that suppressed the output of found configurations. Suppressing the output was needed to avoid the configuration task to become I/O bound due to a large number of atoms printed for each configuration. Instead, just the number of found configurations was reported.

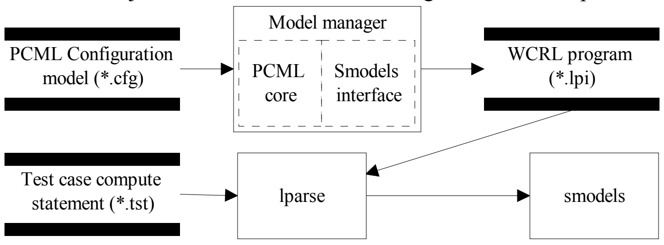


Figure 4. Test setup

For each configuration model, we generated 100 test cases with 2 requirements, 100 test cases with 4 requirements, etc, for each even number of requirements up to the total number of groups. The random requirements in a test case were expressed as a smodels compute statement. If a configuration was found with the requirements of the test case, the test case was considered *satisfiable*, otherwise it was considered *unsatisfiable*.

The tests were run on a laptop PC with 1 GHz Mobile Pentium III processor, 512 MB RAM, and Windows 2000 Professional. All timings were performed using the test system's built-in clock. All times are reported in seconds. A Java based test generator and driver was used to generate and execute the test cases.

The configuration models, test cases, test case run logs, full results as well as the modified Smodels source files and the Windows executable are available at http://soberit.hut.fi/pdmg/empirical_cfg/.

4.4 Results

We briefly explain the measurements before proceeding to the results. *Time to translate PCML to WCRL* includes the time needed by a running Model Manager process to load and translate a PCML configuration model to WCLR and to save the output.

Total duration of a test case includes creating the smodels process for the test case, extracting the number of found answers and the duration reported by smodels, and writing the output log. *Smodels duration* includes the time the smodels executable uses for reading the BCRL program and the time used for computation. *Non smodels time* includes the time to start a test case, run lparse and start smodels, and to gather the results from smodels output (= total duration – smodels duration).

Run time characteristics of the configuration models without the effect of test cases are given in Table 2. The results start with the time to translate PCML to WCRL ("PCML WCRL"), the result is the average of 100 executions. In addition, all configuration models were run once on smodels to find all configurations of each model without any requirements. Table 2 lists the number of configurations ("#Configs") after symmetry breaking, the smodels duration ("Smodels"), as well as the rate of configurations found per second (#Configs/s). "Non smodels" is averaged non smodels time from running the test cases.

Cfg. Model	ESVS	FS	FX	Vehicle
PCML WCRL	8,2 s	8.0 s	5.0 s	1.3 s
#Configs	1,841,356,800	36,106,560	1,136,160	268,800,000
Smodels (s)	18401.4 s	377.1 s	17.0 s	2537.2 s
Configs / s	100066	95748	66833	105944
Non smodels	0,12 s	0,13 s	0,14 s	0,12 s

Table 2. Run time characteristics of configuration models

ESVS		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	89	0,37	189441067	88238	0,30
4	61	0,35	18987439	76849	0,28
6	25	0,34	2234799	72687	0,29
8	9	0,33	211432	19957	0,28
10	4	0,31	1920	263	0,29
12	1	0,32	15552	526	0,29
14-28	0	-	-	-	0,30

Table 3. ESVS compressor results with test cases

FS		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	97	0,33	3583976	84436	0,27
4	76	0,31	706695	74793	0,29
6	55	0,30	90602	55829	0,28
8	26	0,29	10985	20411	0,28
10	15	0,30	1512	4269	0,28
12	10	0,33	719	2173	0,28
14	4	0,30	29	109	0,28
16-22	0	-	-	-	0,28

Table 4. FS compressor results with test cases

FX		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	93	0,23	116646	42200	0,20
4	69	0,22	12558	20116	0,25
6	43	0,21	2537	8552	0,19
8	18	0,21	307	1439	0,19
10	9	0,23	47	181	0,22
12	2	0,27	14	50	0,26
14	1	0,19	5	28	0,20
16	0	-	-	-	0,20

Table 5. FX compressor results with test cases

Vehicle		Find first	Find all		
#req	#sat	(s)	#cfgs / case	#cfgs /s	Unsat (s)
2	95	0,05	37317928	98238	0,04
4	85	0,05	5855831	88913	0,04
6	59	0,05	747205	84642	0,05
8	33	0,05	108638	71394	0,05
10	22	0,09	29136	54358	0,07
12	8	0,07	9526	42361	0,08
14	4	0,06	289	3853	0,08
16	0	-	-	-	0,08
18	2	0,06	9	140	0,07
20-24	0	-	-	-	0,06

Table 6. Vehicle results with test cases

Tables 3 to 6 show our main results from running the generated random test cases. The first run of the test cases evaluated the performance of finding one configuration that satisfies the requirements. The second run evaluated the performance of finding all the configurations that satisfy the requirements. Each row lists the number of requirements ("#req") and the number of satisfiable cases ("#sat"). Note that the sum of satisfiable and unsatisfiable cases is 100. "Find first" gives the average smodels duration of finding one configuration that satisfies the requirements, and "Unsat" gives the average smodels duration to determine unsatisfiability, taken from the second run. "Find all" gives the average number of configurations per satisfiable case ("#cfgs /case") and the average rate of configurations found per second ("#cfgs / s"). Non smodels time from Table 2 can be added to the results to get the average total duration of finding the first configuration or determining unstatisfiability.

The test arrangement caused occasional random delays of approximately ½ second, possibly due to garbage collection in the Java environment, the functions of the operating system or the virus scanner. Therefore maximum durations are not repeatable and only average results are shown. The maximum non-repeatable smodels time for finding one configuration or determining unsatisfiability was still below 0.7 s. Repeatable times were close to the average, typically approximately within 20% of the average, except for the vehicle model, where average duration was always less than 0.1s causing small absolute errors to show major relative differences.

5 DISCUSSION AND PREVIOUS WORK

In this section we first discuss our implementation and empirical results and compare our empirical results to previous work.

Our results indicate performance adequate both for batch mode configuration and interactive configuration with the simple case products. There were no test cases with repeatable significantly inferior performance. Also, there was no significant change of performance as a function of the number of requirements. The average configurations per second results show weakening with increasing number of requirements. However, this seems to be mostly illusory: because the number of configurations with many requirements is small, Smodels duration comes mostly from reading the BCRL program and from setting up the computation.

Our case products were small but we feel that they are representative of what is needed in sales configuration. We expect that the good performance of our configurator can be generalized to many products suitable for web based sales configuration.

No critically constrained problems were found and no phase transition behavior was apparent. As expected, the number of configurations seems to decrease exponentially as the number of requirements increases. Minor exceptions due to random requirements were encountered in the Vehicle and ESVS configuration models.

The case models had no maximum cardinalities larger than one and a component type was usually used as an allowed type only in one part definition. Therefore the significance of symmetry breaking for performance was low. However, it is evident that several forms of symmetries remain unbroken and new important forms arise when implementing the full ontology, e.g. port and connection oriented concepts.

Our approach in individual generation may create more individuals than needed resulting in unnecessarily large compiled models. On the other hand knowing all individuals can make propagation more efficient in smodels and thus enhance performance. This kind of individual generation was also straightforward to implement in conjunction with our compilation strategy.

Running lparse for each test case conflicts with the knowledge compilation principle and our normal way of using the configurator. As the results indicate, running time of lparse for the case products was small. According to our experiences, the time required by lparse to translate a WCRL program increases significantly with large cardinalities.

We measure performance using execution time due to its practical importance for users and its suitability to searching for phase transition behavior. It would be useful to use metrics that are independent of processor power, efficiency of implementation tools and the technology used. Unfortunately such metrics are difficult to define. For example, the number of consistency checks is not commensurate between different technologies such as CSP and logic based approaches. Compromises between propagation and search also significantly affect the number of needed consistency checks.

According to our experiences, the timing result averages are repeatable only to 1/10th of a second. For example, the average time to show unsatisfiability with the Vehicle model differed up to 16 ms between 2 runs making the 1/100th second reading inaccurate. Average times with the FX model varied in one case even by 60 ms between two runs. The repeatability of the results would be improved and maximum durations would become more reliable by executing the test cases several times, excluding the worst results to eliminate the effect of random delays. Fully automatic generation of the test cases from configuration models would make testing easier.

We note that our test methodology can be applied relatively easily to other formalisms like CSP. In CSP, a requirement group could correspond to a CSP variable and a requirement item to one possible assignment for that variable.

We now compare our work briefly with other similar work. Syrjänen configured the main distribution of Debian GNU/Linux using configuration models expressed using an extension of normal logic programs. The configuration task was to select a maximum set of mutually compatible software packages satisfying random user requirements that exclude or include some packages. Average Smodels time for configuration was 1.06 s for Debian 2.0 with 1526 packages and 1.46 s for Debian 2.1 with 2260 packages. The tests were run on a 233 Mhz Intel Pentium II on Linux [4]. The configuration duration was approximately the same as in our largest ESVS model (adjusted for our roughly four times faster processor). Syrjänen's approach seems to perform better than ours as the Debian configuration models are substantially larger.

Sharma and Colomb developed a constraint logic programming (CLP) based language for configuration and diagnosis tasks. Experimental results stem from thin ethernet cabling configuration. The largest 12 node configuration included 126 port connections and required 12 seconds of CPU time on a dual 60 Mhz SuperSparc processor based system to find a configuration [5]. Direct performance comparison to our work is difficult due to port and connection oriented domain, different processor power, and missing details.

Mailharro used the Ilog system to configure the instrumentation and control hardware and software of nuclear power plants. Several thousand component individuals were created and interconnected in about an hour of execution time on a Sun Sparc 20 [8]. The case product is larger and more complex than ours. Direct performance comparison is not possible due to limited details available.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have briefly described a configurator implementation based on mapping its modeling language to weight constraint rules, a form of logic programs. The configurator uses a state-of-the-art general implementation of weight constraint rules for computing configurations satisfying user requirements. Our configurator implementation addressed three issues important for efficiency: off-line compilation of configuration models, limiting the size of a configuration to be finite in a semantically justified way, and breaking symmetries in the set of configurations. However, improved symmetry breaking and generative or more optimal individual generation are subjects for further work.

We then aimed to assess the difficulty of configuration problems as well as the efficiency of our configurator through empirical testing. We defined a relatively modeling-language-independent method for testing configurators based on the idea of simulating a naïve user inputting random requirements to a configurator. The methodology enables systematic testing using a small set of products.

We modeled four products taken from two domains from a sales configuration point of view. The modeling language of the configurator was found adequate for modeling these products.

The empirical results indicate that our configurator is efficient enough for sales configuration use. The results also support the common wisdom that configuration problems are relatively easy to solve. However, our small sample of relatively small sales configuration models originates from two domains only. Thus, more tests are needed with larger and potentially more difficult to configure products taken from different domains (e.g. telecommunications and electronics), and modeled from engineering point of view.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from Technology Development Centre of Finland (Tekes). The work of the third author has been supported by the Academy of Finland (project 53695). We thank Hannu Peltonen for authoring most of the first version of the PCML specification. We thank him, Andreas Anderson and Jan Elfström for implementing significant parts of the configurator. We also thank Tommi Syrjänen and Patrik Simons for developing Smodels with configuration in mind. Finally, we wish to thank Gardner Denver Oy for sharing the configuration knowledge.

REFERENCES

[1] A. Felfernig, G. Friedrich and D. Jannach, 'UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems', *International Journal of Software Engineering and Knowledge Engineering*, **10**, 449-469, 2000.

[2] T. Soininen, I. Niemelä, J. Tiihonen and R. Sulonen, Representing Configuration Knowledge With Weight Constraint Rules. *In Proc. of the AAAI Spring 2001 Symposium on Answer Set Programming*, 2001.

[3] S. Mittal and B. Falkenhainer, Dynamic Constraint Satisfaction Problems, *in Proc. of the 8th National Conf. on AI (AAAI-90)*, 25-32, 1990.

[4] T. Syrjänen, Including Diagnostic Information in Configuration Models, i*n Proc. of the First International Conference on Computational Logic, (Volume 1861 of Lecture Notes in Artificial Intelligence)*, 2000.

[5] N. Sharma and R. Colomb, Mechanising Shared Configuration and Diagnosis Theories Through Constraint Logic Programming, *Journal of Logic Programming* **37**, 255-283, 1998.

[6] M. Stumptner, G. Friedrich, A. Haselböck, Generative constraint-based configuration of large technical systems. *AI EDAM,* **12**, 307-320, 1998.

[7] D. McGuinness and J. Wright, An Industrial-strength Description Logic-Based Configurator Platform, *IEEE Intelligent Systems & Their Applications,* **13**, 69-77, 1998.

[8] D. Mailharro, A classification and constraint-based framework for Configuration, *AI EDAM*, **12**, 1998.

[9] Mackworth A.K. Consistency in Networks of Relations. *Artificial Intelligence,* **8**, 99-118, 1977.

[10] T. Soininen, E. Gelle, and I. Niemelä, A Fixpoint Definition of Dynamic Constraint Satisfaction. *In Principles and Practice of Constraint Programming - CP'99 (LNCS 1713)*, 419-433, 1999.

[11] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, Towards a General Ontology of Configuration, *AI EDAM,* **12**, 357–372, 1998.

[12] P. Simons, I. Niemelä, and T. Soininen, Extending and implementing the stable model semantics. *To appear in Artificial Intelligence, Special Issue of Knowledge Representation and Logic Programming.*

Knowledge Compilation for Product Configuration

Carsten Sinz[1]

Abstract. In this paper we address the application of knowledge compilation techniques to product configuration problems. We argument that both the process of generating valid configurations, as well as validation of product configuration knowledge bases, can potentially be accelerated by compiling the instance independent part of the knowledge base. Besides giving transformations of both tasks into logical entailment problems, we give a short summary on knowledge compilation techniques, and present a new algorithm for computing unit-resolution complete knowledge bases.

1 Introduction

Configuration of complex products is a computation intensive task. In most formalisms proposed in the literature [5, 10, 11], generating a consistent configuration can be intractable in the worst case, and is at best an NP-hard problem. Moreover, in a typical setting, huge series of configuration runs have to be performed for the same kind of product, but different customer demands. The closely related task of checking a product configuration knowledge-base for consistency [7] exhibits similar characteristics. Here a large number of validation properties have to be checked for a given knowledge-base.

These conditions make it attractive to investigate the application of knowledge compilation techniques (see Cadoli and Donini's article for a survey on knowledge compilation [2]). For a set of common problem instances, knowledge compilation separates the computational task into an instance dependent and an instance independent part. The latter can be solved in advance during a preprocessing step, which can potentially lead to a speedup in the overall run time.

A significant advantage of pre-compiled knowledge-bases is that in the lucky case of successful compilation into a knowledge base of reasonable size, short runtimes can be guaranteed for all individual configuration processes.

2 Formalisms for Product Configuration

In this paper, we consider two formalisms for product configuration. We use them as representatives for demonstrating applicability of knowledge compilation for both generating valid configurations and checking consistency of knowledge bases.

The first formalism is a slight variant of the logical theory of configuration presented by Felfernig *et al.* [5], which complies with Mittal and Frayman's component-port representation for configuration knowledge [11]. The second is a simplified version of the formalism used for the validation of DaimlerChrysler's engineering and production configuration system [7].

In Felfernig's system, a configuration problem consists of a domain description $\mathcal{D}$ and a system requirements specification $\mathcal{S}$, from which a consistent (*valid*) configuration c has to be constructed[2]. The domain description is a set of predicate logic sentences expressing compatibility constraints on the product's parts, and additional axioms describing and restricting the constraints language; the system requirements specification states customer demands on the desired product, again in the form of a set of predicate logic sentences. Then, a conjunction c of ground literals is a solution to the configuration problem $\langle \mathcal{D}, \mathcal{S} \rangle$, or a *valid* configuration, when it is logically consistent with the domain description and the requirements specification, i.e., $\mathcal{D} \cup \mathcal{S} \cup \{c\} \not\models \bot$. For our purpose we consider the propositional variant of this formalism. We assume a finite universe, or a universe with a finite number of equivalence classes with respect to the domain description and the requirements specification. Now the propositional case can be obtained from the first-order case by replacing all sentences of $\mathcal{D}$ and $\mathcal{S}$ by a conjunction of their ground instances, similar to a Herbrand expansion.

The second formalism we consider consists of a set of propositional constraint rules that make up a knowledge-base describing valid products [7]. The semantics of the whole knowledge-base can be interpreted as a propositional formula $\mathcal{B}$ whose models are the valid configurations. To check its consistency, we generate a set D of validation properties d_i and test whether or not the knowledge-base satisfies them. Therefore, we have to determine whether $\mathcal{B} \models d_i$ for all $d_i \in D$.

3 Logical Entailment and Knowledge Compilation

Problem instances of both formalisms can be formulated as *propositional entailment* problems, where the question is to find the deducible consequences a of a theory $\mathcal{T}$ (a set of propositional sentences). For the purpose of knowledge compilation, we partition the theory $\mathcal{T}$ into a constant part $\mathcal{T}_\text{c}$ and a varying part $\mathcal{T}_\text{v}$. The constant part $\mathcal{T}_\text{c}$ is then replaced by an equivalent, but computationally preferable, compiled theory $\mathcal{T}_\text{c}'$, and the varying part of the theory is moved to the consequence by means of the deduction theorem. Thus, the general entailment problem

$$\mathcal{T}_\text{c} \cup \mathcal{T}_\text{v} \models a$$

is restated in the compiled theory as

$$\mathcal{T}_\text{c}' \models a \vee \bigvee_{s \in \mathcal{T}_\text{v}} \neg s \ . \tag{1}$$

[1] Symbolic Computation Group, WSI for Computer Science, University of Tübingen, Sand 13, 72076 Tübingen, Germany

[2] Ferfernig *et al.* [5] distinguish between consistent and valid configurations, where valid configurations have to fulfill additional completeness axioms. In this article we always refer to the augmented version with completeness axioms added when talking about valid or consistent configurations.

This transformation especially offers advantages when (a) the constant part of the theory is large compared to the variable part of the theory and the consequence to be checked, (b) the compilation of theory $\mathcal{T}_c$ into $\mathcal{T}_c'$ is efficient, and (c) there is a large number of entailment checks to be performed.

In transforming the first formalism, the fixed part $\mathcal{T}_c$ is the domain description $\mathcal{D}$, and the varying part is the systems requirement specification $\mathcal{S}$. The task now is to find a valid configuration consistent with $\mathcal{D}$ and $\mathcal{S}$ by a series of entailment checks. When we consider the inverted requirements specification $I = \bigvee_{s \in \mathcal{S}} \neg s$ being represented in conjunctive normal form (CNF), i.e. $I = i_1 \wedge \cdots \wedge i_k$, we can perform k entailment checks $\mathcal{D}' \models i_j$ for $1 \leq j \leq k$ within the compiled theory $\mathcal{D}'$ to find configurations conforming with the domain description: For each i_j with $\mathcal{D}' \not\models i_j$ the conjunction of literals $\neg i_j = l_1 \wedge \cdots \wedge l_m$ is a minimal valid configuration. This can be verified by observing that the configuration $c = \neg i_j = l_1 \wedge \cdots \wedge l_m$ is consistent with $\mathcal{D} \cup \mathcal{S}$. As $\mathcal{D}' \not\models i_j$, and $\mathcal{D}'$ is logically equivalent to $\mathcal{D}$, we have $\mathcal{D} \cup \{\neg i_j\} \not\models \bot$, hence a fortiori $\mathcal{D} \cup \{\neg i_1 \vee \cdots \vee \neg i_k\} \not\models \bot$. Now as S is equivalent to $\neg I = \neg i_1 \vee \cdots \vee \neg i_k$ and $c = \neg i_j$ for some j, we obtain $\mathcal{D} \cup \mathcal{S} \cup \{c\} \not\models \bot$.

Further non-minimal valid configurations can be generated using the following scheme: for each literal l not occurring in i_j we test whether $\mathcal{D}' \not\models i_j \vee \neg l$. If this is the case, we can safely add l to the configuration without violating validity.

In summary, to treat the transformation of the first formalism, we set $a = \bot$ in Formula 1, and consider a special representation (DNF) of the variable part of the theory T_v.

For transformation of the second formalism (checking validation properties) we assume the validation properties d_i to be in conjunctive normal form, i.e. $d_i = d_{i,1} \wedge \cdots \wedge d_{i,k}$. Then, after setting the constant part $\mathcal{T}_c$ of the theory to $\mathcal{T}_c = \{\mathcal{B}\}$, and noting that there is no variable part, i.e. $\mathcal{T}_v = \emptyset$, the test for property d_i decomposes into k entailment checks $\mathcal{T}_c' \models d_{i,j}$ in the compiled theory $\mathcal{T}_c'$.

In both formalisms we only have a restricted form of entailment checks, namely only tests $\mathcal{T}' \models a$, where a is a clause. Therefore we restrict our attention to propositional clausal entailment in the following. Knowledge compilation aims at generating theories for which the entailment problem is tractable—decidable in polynomial time—whereas the general propositional clausal entailment problem is coNP-complete [2].

4 Knowledge Compilation Techniques

Knowledge compilation and concequence finding are active areas of research [2, 9, 13]. The methods proposed in the literature are usually separated into two main categories: *approximate* and *exact* compilations. Approximate compilation mostly appears in the form of theory approximation, where a theory $\mathcal{T}$ is approximated by a computationally more tractable theory $\mathcal{T}'$. Selman and Kautz [13] use two approximating Horn theories, one approximating from above ($\mathcal{T} \models \mathcal{T}_{\mathrm{ub}}$) and the other from below ($\mathcal{T}_{\mathrm{lb}} \models \mathcal{T}$).[3] To decide entailment for a clause c, algorithm `THEORY-APPROX`, as shown in Figure 1, is employed. Entailment for Horn theories, i.e. for $\mathcal{T}_{\mathrm{ub}}$ and $\mathcal{T}_{\mathrm{lb}}$, can be decided in linear time, so algorithm `THEORY-APPROX` is supposed to decide many cases efficiently: the better the approximation, the more cases are computationally tractable. However, computation of good Horn approximations can be quite hard (computation of best approximations is NP-hard). Selman and Kautz [13] present different algorithms for their computation.

```
ALGORITHM THEORY-APPROX
INPUT:    $\mathcal{T}_{\mathrm{lb}}, \mathcal{T}, \mathcal{T}_{\mathrm{ub}}, c$ with $\mathcal{T}_{\mathrm{lb}} \models \mathcal{T} \models \mathcal{T}_{\mathrm{ub}}$
OUTPUT:   $true$ if $\mathcal{T} \models c$, $false$ otherwise
BEGIN
    IF $\mathcal{T}_{\mathrm{ub}} \models c$ THEN return $true$
    ELSE IF $\mathcal{T}_{\mathrm{lb}} \not\models c$ THEN return $false$
    ELSE return $\mathcal{T} \models c$
END
```

Figure 1. Theory Approximation Algorithm.

Exact compilation methods, as opposed to approximations, try to find a theory $\mathcal{T}'$ that is equivalent to $\mathcal{T}$, for which the entailment problem is tractable, i.e., decidable in polynomial time. The predominant method for computing such compilations is by generating prime implicants or prime implicates of the original theory.

In the following, we consider a compilation by the computation of prime implicates. The source theory then usually also is in conjunctive normal form (CNF)—a common incidence in practice. An *implicate* c of a theory $\mathcal{T}$ is a non-trivial clause (without complementary literals) such that $\mathcal{T} \models c$; moreover, c is a *prime implicate* if no proper sub-clause[4] of c is also an implicate of $\mathcal{T}$.

Computing the set $\mathrm{PI}(\mathcal{T})$ of all prime implicates of a theory $\mathcal{T}$ yields a theory $\mathcal{T}'$ that is equivalent to $\mathcal{T}$ and has the following important property: a clause c is a consequence of $\mathcal{T}$, i.e. $\mathcal{T} \models c$, iff there is a clause $p \in \mathrm{PI}(\mathcal{T})$ that is a sub-clause of c. Thus, using $\mathcal{T}' = \mathrm{PI}(\mathcal{T})$ as a compiled version of the theory $\mathcal{T}$, clausal entailment for a clause a can be decided in linear time in the size of $\mathcal{T}'$ and the query a.

Different algorithms have been proposed to compute the set of prime implicates of a theory [3, 12, 15]. However, the number of prime implicates may be exponential in the size of the theory $\mathcal{T}$, and therefore different strategies have been developed to compute more compact exact compilations. Among the extensions are computations of prime implicates from no-merge resolvents [4], theory prime implicates [8], and tractable cover compilations [1]. In the following, we will describe del Val's work [4] in more detail.

The prime implicate computation from no-merge resolvents is sometimes also referred to as unit-resolution (UR) complete compilation, and the idea is to delete those prime implicates from $\mathrm{PI}(\mathcal{T})$ that can be derived by a UR refutation proof from the remaining theory. As UR refutations can be computed in linear time, this means a shift from precompiled knowledge to deduction by a calculus that is still tractable. In the following we denote UR derivations by $\vdash_{\mathrm{u}}$, and use the convention that for a clause $c = l_1 \vee \cdots \vee l_k$ the notation $\bar{c}$ stands for the set of units $\{\neg l_1, \ldots, \neg l_k\}$ obtained by negating c. Our goal now is to compute a set $\mathcal{T}^*$ of clauses that is equivalent to $\mathcal{T}$, and for which

$$\mathcal{T}^* \cup \bar{c} \vdash_{\mathrm{u}} \bot$$

holds for all clauses c with $\mathcal{T} \models c$. Then all consequences of $\mathcal{T}$ can be derived by UR refutations from $\mathcal{T}^*$. Of course, setting $\mathcal{T}^* = \mathrm{PI}(\mathcal{T})$ would be a solution, but this is often not practical and does not

[3] The bounds are defined in terms of models: the lower bound has fewer, the upper bound more models than the given theory.

[4] Clause c is a (proper) sub-clause of d if the set of literals of c is a (proper) subset of the literals of d.

deliver the best, i.e. minimal, theory. Del Val [4] suggests different candidates for theory $\mathcal{T}^*$.

We now want to derive a precise characterization of an optimal $\mathcal{T}^*$, expressed by means of a fixed-point equation. Therefore, we define $\mathcal{T}^*_{\text{opt}}$, an optimal solution for UR complete compilation, as a smallest (regarding set inclusion) solution $\hat{\mathcal{T}}$ of the formula

$$\forall c \in \text{PI}(\mathcal{T})\big(c \in \hat{\mathcal{T}} \iff \bar{c} \cup \hat{\mathcal{T}} \setminus \{c\} \not\vdash_{\text{u}} \bot\big) \ . \tag{2}$$

Then a clause c from the set of prime implicates of theory $\mathcal{T}$ is in the solution theory $\mathcal{T}^*$ iff it cannot be derived by a UR refutation from $\mathcal{T}^*$ without c.

5 An Alternative Algorithm to Compute UR Complete Knowledge-Bases

Based on Formula 2 we now give an algorithm for UR complete knowledge compilation. Our algorithm, as well as all of del Val's algorithms for computation of UR complete compilations, is based on prime implicate generation. This has the advantage that advances in prime implicate computation can also improve knowledge compilation, but suffers from the drawback that in the worst case an exponential number of clauses has to be generated, even if the final result does not show this exponential blow-up. Our alternative algorithm, as shown in Figure 2, computes a different, in some cases smaller, compiled knowledge-base than del Vals algorithms.

```
(1)  ALGORITHM UR-COMPILATION
(2)  INPUT: $\mathcal{T}$
(3)  OUTPUT: $\mathcal{T}^*$, which is a solution to Formula 2
(4)  BEGIN
(5)    $\mathcal{T}^*$ := PI($\mathcal{T}$);
(6)    FOR EACH $c \in \mathcal{T}^*$ DO
(7)      IF $\bar{c} \cup \mathcal{T}^* \setminus \{c\} \vdash_{\text{u}} \bot$ THEN
(8)        $\mathcal{T}^*$ := $\mathcal{T}^* \setminus \{c\}$;
(9)        compute minimal $D(c) \subseteq \mathcal{T}^*$ with
(10)         $\bar{c} \cup D(c) \vdash_{\text{u}} \bot$;
(11)       FOR EACH $c' \in \text{PI}(\mathcal{T}) \setminus \mathcal{T}^*$ with $c \in D(c')$ DO
(12)         IF $\bar{c}' \cup \mathcal{T}^* \vdash_{\text{u}} \bot$ THEN
(13)           update $D(c')$
(14)         ELSE
(15)           $\mathcal{T}^*$ := $\mathcal{T}^* \cup \{c'\}$;
(16) END
```

Figure 2. Algorithm for UR Complete Knowledge Compilation.

Starting with the whole set of prime implicates, clauses are successively temporarily removed (line 8) from the result set $\mathcal{T}^*$, if the entailment of a clause c can also be obtained from $\mathcal{T}^*$ without c by a UR refutation (line 7). Then a justification $D(c)$ of why the deletion of c was possible is computed (lines 9/10). This justification contains the clauses involved in a shortest UR refutation of c. By removing clause c, UR refutation proofs of other, already removed clauses, may break. So for each previously removed clause c' it is checked whether a UR refutation proof of c' is still possible (lines 11/12). If this is the case, the proof, i.e. the justification $D(c')$, is updated (line 13) analogous to the computation in lines 9/10. Otherwise the formerly removed clause c' is re-added to the working theory $\mathcal{T}^*$ (line 15), and the whole process is repeated until no further changes result, and thus a fixed point is reached.

The main loop of the algorithm may be interrupted after each round, and still returns a UR complete theory equivalent to the input theory, yet not necessarily minimal.

To further illustrate the effects of our algorithm let us consider an example[5]. Let

$$\mathcal{T} = \{\ pqs, p\bar{q}r, \bar{p}qt, \bar{p}\bar{q}u, \bar{t}vw, \bar{v}x\ \}\ .$$

Computation of the set of prime implicates in line 5 of the algorithm then yields

$$\begin{aligned}\mathcal{T}^* = \{\ & pqs, p\bar{q}r, \bar{p}qt, \bar{p}\bar{q}u, \bar{t}vw, \bar{v}x, qst, \bar{q}ru, prs,\\ & \bar{p}tu, rstu, \bar{p}qvw, qsvw, \bar{p}uvw, rsuvw, \bar{t}wx,\\ & \bar{p}qwx, qswx, \bar{p}uwx, rsuwx\ \}\ .\end{aligned}$$

Suppose we first choose $c = rsuwx$ in line 6. Then we have

$$\{\bar{r}, \bar{s}, \bar{u}, \bar{w}, \bar{x}\} \cup \mathcal{T}^* \setminus \{rsuwx\} \vdash_{\text{u}} \bot,$$

and we thus remove c from $\mathcal{T}^*$. A minimal justification $D(c)$ for deleting c is

$$D(c) = \{\ pqs, p\bar{q}r, \bar{p}uwx\ \}\ .$$

Selecting $c = \bar{p}uwx$ next, we obtain $\{p, \bar{u}, \bar{w}, \bar{x}\} \cup \mathcal{T}^* \setminus \{\bar{p}uwx\} \vdash_{\text{u}} \bot$, and therefore we also remove this clause and compute $D(c)$ for it. But now we have for $c' = \{rsuwx\}$ that $c' \in \text{PI}(\mathcal{T}) \setminus \mathcal{T}^*$ and $c \in D(c')$. As $\{\bar{r}, \bar{s}, \bar{u}, \bar{w}, \bar{x}\} \cup \mathcal{T}^* \vdash_{\text{u}} \bot$ still holds, we just have to update the justification, e.g. $D(c') = \{rsuvw, \bar{v}x\}$. Repeating the algorithm's steps over and over we reach the fix-point

$$\mathcal{T}^* = \{\ pqs, p\bar{q}r, \bar{p}qt, \bar{p}\bar{q}u, \bar{t}vw, \bar{v}x, \bar{q}ru, prs, \bar{t}wx\ \}\ .$$

The complexity of our algorithm is dominated by the prime implicate computation step in line 5, which—as is well known—may in the worst case require exponential space (and thus time) (see, e.g., [9]). It remains an interesting task to find an algorithm for UR complete knowledge-base compilation that is not based on prime implicate computation.

6 Preliminary Experimental Results

We are currently starting experiments with our algorithm on databases from automotive product configuration. First results obtained from a prototypical implementation of our algorithm are shown in Table 1.

problem	$\lvert\text{PS}\rvert$	$\lvert\mathcal{T}\rvert$	$\lvert\text{PI}(\mathcal{T})\rvert$	$\lvert\mathcal{T}^*\rvert$
Adder	21	50	9,700	1,183
C250FV	1,465	2,356	2,492	1,837
C210FVF	1,934	3,985	496,050,800	–

Table 1. Experimental Results

The first example is taken from Forbus and de Kleer's book [6], which contains databases that are often used as benchmarks in the knowledge compilation community. The following two examples are knowledge bases describing valid model lines of DaimlerChrysler's

[5] This is Example 1 from del Val [4]. We also use his abbreviated notation for clauses, e.g. writing $pq\bar{r}$ instead of $p \vee q \vee \neg r$.

Mercedes cars. These databases are used by Küchlin and Sinz [7] for validation checks.

The columns of Table 1 show in turn the number of propositional variables, the size of the database in number of clauses, the number of prime implicates of the theory, and the number of prime implicates that remain after application of our algorithm.

problem	t_{PI}	t_{reduce}	t_{UR}
Adder	0.38	2683.10	2683.48
C250FV	93.46	220.42	313.88
C210FVF	426.55	–	–

Table 2. Compilation Times

In Table 2 we present runtimes for our UR complete knowledge compilation algorithm. The last column shows the total runtime t_{UR}, which is split into the time for prime implicate computation (t_{PI}) and reduction of the prime implicate set in algorithm lines 6-14 (t_{reduce}). We used Simon and del Val's BDD-based implementation *zres* as a prime implicate generator [14], and ran it on a Pentium III running at 733 MHz. For the reduction part we used an experimental implementation written in Haskell, compiled with the Glasgow Haskell Compiler, version 4.04. Our implementation failed on reducing the prime implicates for the C210FVF data set, which is indicated by a dash in the tables. We are currently working on an implementation in C++ using more efficient data structures.

7 Conclusion and Future Work

We presented a method to compile the fixed part of product configuration databases, and proposed a new algorithm for the computation of UR complete compilations. First experiments indicate that, at least for validation of configuration database properties, our method is applicable.

Besides conducting further experiments, we consider improvement of knowledge compilation algorithms, e.g. by developing exact algorithms that do not require a prior computation of all prime implicates, as a promising area of future research. Moreover, it could be of interest to evaluate the performance of knowledge compilation on other product documentation formalisms and for other practical application areas of configuration.

ACKNOWLEDGEMENTS

I am indebted to Laurent Simon for providing me his *zres* implementation for prime implicate generation.

REFERENCES

[1] Y. Boufkhad, G. Éric, P. Marquis, B. Mazure, and S. Lakhdar, 'Tractable cover compilations', in *Proc. of the 15th Intl. Joint Conf. on Artificial Intelligence (IJCAI'97)*, pp. 122–127, Nagoya, Japan, (August 1997).

[2] M. Cadoli and F.M. Donini, 'A survey on knowledge compilation', *AI Communications*, **10**(3–4), 137–150, (1997).

[3] O. Coudert and J.C. Madre, 'Implicit and incremental computation of primes and essential primes of boolean functions', in *Proc. of the 29th Design Automation Conf. (DAC 1992)*, pp. 36–39, Anaheim, CA, (June 1992).

[4] A. del Val, 'Tractable databases: How to make propositional unit propagation complete through compilation', in *Proc. of the 4th Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR'94)*, pp. 551–561, Bonn, Germany, (May 1994).

[5] A. Felfernig, G.E. Friedrich, D. Jannach, and M. Stumptner, 'Consistency-based diagnosis of configuration knowledge bases', in *Proc. of the 14th European Conf. on Artificial Intelligence (ECAI 2000)*, pp. 146–150, Berlin, Germany, (August 2000).

[6] K.D. Forbus and J. de Kleer, *Building Problem Solvers*, MIT Press, 1993.

[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] P. Marquis, 'Knowledge compilation using theory prime implicates', in *Proc. of the 14th Intl. Joint Conf. on Artificial Intelligence (IJCAI'95)*, pp. 837–845, Montréal, Canada, (August 1995).

[9] P. Marquis, 'Consequence finding algorithms', in *Handbook of Defeasable Reasoning and Uncertainty Management Systems*, eds., D.M. Gabbay and Ph. Smets, volume 5, 41–145, Kluwer, (2000).

[10] D.L. McGuinness and J.R. Wright, 'Conceptual modelling for configuration: A description logic-based approach', *AIEDAM*, **12**(4), 333–344, (1998).

[11] S. Mittal and F. Frayman, 'Towards a generic model of configuration tasks', in *Proc. of the 11th Intl. Joint Conf. on Artificial Intelligence*, pp. 1395–1401, Detroit, MI, (August 1989).

[12] I. Rish and R. Dechter, 'Resolution versus search: Two strategies for SAT', *J. Automated Reasoning*, **24**(1–2), 225–275, (February 2000).

[13] B. Selman and H. Kautz, 'Knowledge compilation and theory approximation', *JACM*, **43**(2), 193–224, (1994).

[14] L. Simon and A. del Val, 'Efficient consequence finding', in *Proc. of the 17th Intl. Joint Conf. on Artificial Intelligence (IJCAI'01)*, pp. 359–365, Seattle, WA, (August 2001).

[15] P. Tison, 'Generalized consensus theory and application to the minimization of boolean functions', *IEEE Transactions on Electronic Computers*, **EC-16**(4), (August 1967).

Ideas for Removing Constraint Violations with Heuristic Repair

Gottfried Schenner[1] and Andreas Falkner[2]

Abstract. This short paper describes our current results with using heuristic repair methods in combination with an object-oriented constraint based configurator. There are many sources of constraint violations in constraint based configuration applications. They can be caused by an unfinished configuration process, a new (stricter) version of the knowledge base which makes existing configurations inconsistent, or a manual modification which is not supported by the knowledge base, but still works in practice. While some configurators force the user to remove inconsistencies manually or prohibit contradictions in the first place, we use heuristic repair methods to support the user by suggesting possible repair steps to resolve the inconsistencies. Whether this approach is also suitable for standalone solving, i.e. resolving constraint violations without user interactions, is a topic for ongoing research.

1 INTRODUCTION

In real world configurator applications, product definitions and user requirements change frequently. For example, new components become available, parts run out of production, new features are required, or constraints for building the product change. The knowledge base of the configurator has to change accordingly [1]. If the configurator loads an old configuration with the new version of the knowledge base, it may detect constraint violations because new constraints were added to the knowledge base. Typically, there are different ways to resolve a contradiction and in most cases only the user knows which one is the best.

A similar situation occurs if we see the configuration process as a sequence of user actions and configurator solving steps: The constraint solver is switched off, i.e. it only reports constraint violations but does not try to repair them. The user manually changes the configuration. If the task he wants to accomplish involves several steps, he will not be able to finish it with one interaction. The configuration is somehow incomplete, thus violating some constraint.

In both situations a configuration violates some constraints and the user may need assistance for treating the violations, i.e. repairing the configuration. This can be done by a solver component that suggests possible solution steps to the user and sorts the solutions by their probability for solving the contradictions.

A different situation arises when the user manually creates a configuration which still works in practice, but does violate some constraints of the knowledge base, either because the knowledge base is faulty or this configuration is a very special case not supported by the knowledge base. Here the user should be able to "allow" the violated constraints, thus telling the solver not to attempt to repair these constraint violations.

The next sections describe the architecture of the configurator, our heuristic strategies for repair, experimental results, and conclusions.

2 SYSTEM OVERVIEW

The configurator for which we are developing our heuristic repair method has been in operation for over a year, and we also have experience with a project engineering tool for the same customer, which has been in everyday use for over 10 years.

The application domain is configuration of the hardware (racks, frames, modules, cables, wrapping etc.) and the software (logical view of the topology, configuration of the runtime software) of railway interlocking stations. Typically the configuration tasks are complex, configuration products are in operation for many years and configurator knowledge bases change during that time due to new or changed features and component types. [2] gives a short description of the domain.

Our configurator system is based on a Java implementation of the COCOS [3] configuration framework. We use an object-oriented database for storing configurations, however this is not a precondition for the ideas in this paper. The knowledge base (class model, constraints) is implemented in Java as well. This approach allows a seamless integration of the configuration engine (COCOS) into the configurator (a standard Java application). For the configurator application the constraint solver is just another process that manipulates the configuration that is an instantiation of the knowledge base.

The set of possible configurations is determined by the possible instantiations of the class model. Each possible configuration can be manually configured by the user with a generic GUI. A consistent configuration is a possible configuration (based on the class model) that does not violate any constraint.

[1] Siemens AG Oesterreich, Program and System Engineering, Erdberger Laende 26, A-1030 Vienna, Austria, email: gottfried.schenner@siemens.com
[2] Siemens AG Oesterreich, Program and System Engineering, Erdberger Laende 26, A-1030 Vienna, Austria, email: andreas.a.falkner@siemens.com

One key feature of the system, which is also essential for the repair process, is the ability to track the variables (attributes, associations) that are referenced in constraints. With these dependencies it is possible to identify the parts of the configuration that need reconfiguration.

3 HEURISTIC REPAIR

Given an inconsistent configuration our basic heuristic repair method repeats the following operations:

1) Based on the currently violated constraints choose a set of possible repair steps. (They can be simple repair steps affecting only one attribute/association or domain specific steps which change larger portions of the configuration.)
2) From these steps, select the most promising one based on the current heuristic. (This can involve a one step lookahead or letting the user decide which step to take.)
3) If an end criterion is met, then stop.

This method can be controlled by various strategies which are often influenced by the domain the solver is operating in. Hardware configuration needs other strategies than software configuration. Typically, hardware components and their replacement are much more expensive in the real world than software changes. Possible heuristics are among others choosing the repair-steps which minimize the overall number of violated constraint expressions or maximize the number of repaired constraint expressions.

Simple repair steps

Intuitively a repair step is a change to the configuration which removes at least one inconsistency. For boolean expressions this may consist in “flipping” the truth value of a boolean variable in order to satisfy the affected boolean expression. I.e. in the case of clauses this corresponds to local hill-climbing procedures [4].

Repairing other types of attributes is a similar procedure. To satisfy the expression preferredBranch.isFrom(“LEFT”,”RIGHT”), preferredBranch being a String variable with the Value “L”, preferredBranch has to be set to one of {“LEFT”,”RIGHT”}.

Attributes also have a state which indicates “who” set the attribute. Attributes with state “USERDEF” were set manually by the user and must not be changed by the solver.

Repairing expressions that involve associations is more complex. The following example shows how a solver can repair an inconsistent configuration by changing associations.

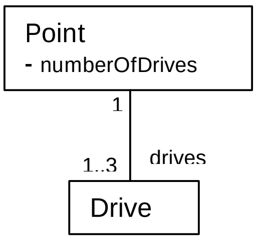


Figure 1

Suppose that a new constraint was introduced for Points (railroad switches) - see UML diagram Figure 1- which states that the number of Drives associated with a Point equals the value of the attribute numberOfDrives.

Constraint:
jcosEq(p.getVar(numberOfDrives),jcosCard(getVar(drives))

Assume that there has been an existing configuration with one Point, with numberOfDrives set to 2, and one associated Drive.

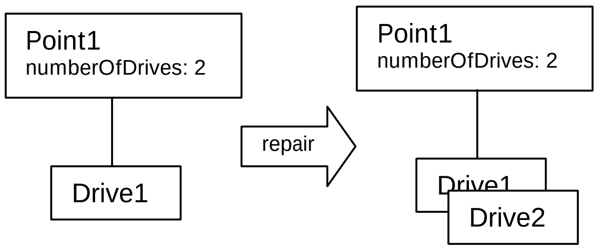


Figure 2

In this situation the newly introduced constraint is violated. There are two possible repair steps to resolve the inconsistency.

1) Associate an additional drive with the given Point. Afterwards the cardinality of the point-drive association equals the numberOfDrives (Figure 2)
2) Set the numberOfDrive attribute to 1, i.e. to the number of currently associated Drives.

In the case of 1) there are further choices. Shall a new Drive be created or an existing one reused ? The second repair step is obviously the cheapest one, but is only available if the value of the numberOfDrives attribute was not set by the user.

This example shows some of the repair-steps possible for associations. If a cardinality constraint for associations is violated then add or remove an element from the association until the cardinality constraint is satisfied. When adding an element, we must decide if there are elements in the configuration which can be used (less costly) or if a new object should be created (more costly) and associated. Note that an Object o1 can only be assigned to an Object o2, if associating it with o2 does not violate the cardinality constraints of its associations.

Since single repair steps only change small parts of the configuration, this approach is especially useful for reconfiguration, i.e. when an already existing system has to be changed to fulfill new requirements.

So far these repair steps where independent of the domain where they are used. To use them in practice the solver has to use a domain dependent solving/repair strategy that defines the costs for removing/creating certain parts and the parts that must not be changed. This reflects the knowledge of a human expert doing the reconfiguration.

Domain specific repair steps

There are some domains where very specific constraints exist. For these domains it may be useful to specify special repairs steps. The following example comes from the domain of configuring a topological view of the interlocking system.

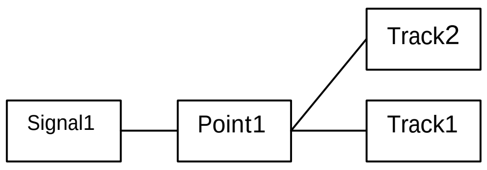


Figure 3

The situation in Figure 3 shows a simple configuration. Assume the user has created this configuration without considering the constraint that after a signal there must be an element of type EOR (end of route). This is expressed with the domain specific constraint expression afterElement(Signal1,EOR).

To repair that inconsistency a number of domain specific actions must be taken. First a new element of type EOR must be created and it must be connected to the appropriate connectors. The result of the repair process is shown in Figure 4.

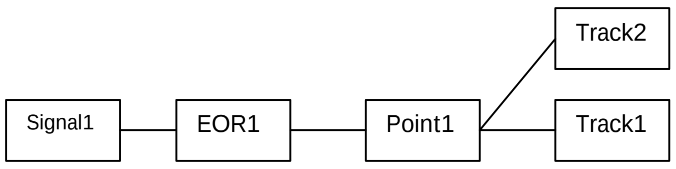


Figure 4

The repair action for the constraint is realized by a method of the constraint class. By this it is encapsulated inside the constraint and can be tested together with the constraint. Knowledge engineers can use these domain specific constraints like any other constraint. It is usually hard to achieve this kind of behavior without using domain specific extensions.

User guided repair

To integrate the repair process into the user interface, the user can select a violated constraint on the GUI and choose an action from a list of possible repair actions. This way the user not only gets an explanation for the constraint violation (the list of dependent variables) but also different possibilities to remove the violation. The user actions are then driven by the set of violated constraints. This also reduces the demand for the implementation of domain specific user interfaces.

The same mechanisms used to repair configurations can also be used to build configurations from scratch. The solving process is then guided by the constraints yet to be satisfied.

4 EXPERIMENTAL RESULTS

We are currently experimenting with various repair strategies. Our goal is to use the repair approach also for stand-alone solving. As our approach is a greedy local search strategy, we experienced all the problems involved with nonsystematic approaches [5] (termination, local maximum, etc.) Often this can be handled by using the appropriate solving strategy or querying the user. Unfortunately the development of a suitable solving strategy is not straight-forward. This is a topic for further research.

We found that if we let the user choose which repair step to take next, he is often capable to finish a configuration without taking back an earlier decision. This is because his decisions are guided by his common sense knowledge about the involved physical objects. For example in HW configuration there are often soft constraints like the demand to fill a frame with modules from left to right. Although any other order would also lead to a working system, this user demand should be fulfilled by the solver. One of the challenges we are facing is to capture that kind of knowledge by providing the right heuristics to the solving process.

5 CONCLUSION

Heuristic repair methods can be successfully used for the task of configuration. They provide a means to treat constraint violations in a uniform manner, independent of the different source of the constraint violations (e.g. changes in the knowledge base, changes of the configuration).

Our main motivation for this work was to support the user during the configuration process. We also introduced domain specific repair steps in order to reduce the number of manual user interactions.

We are now trying to develop a full stand alone solving framework based upon this approach, which is less important than repairing contradictions. Especially when the user changes configurations which have already been deployed to the real world, there is always the need to query the user whether he really wants to change certain parts of the configuration.

This heuristic repair framework can be useful for non-configuration software projects as well. Since we are not using configuration specific extensions in our class model, any systems which use a standard class model [6] can benefit from using constraints for checking valid instances of this class model and using repair methods for fixing invalid ones.

6 REFERENCES

[1] Andreas Falkner, Versioning of Knowledge Bases and Configurations, AAAI 1999 Workshop on Configuration, Orlando, FL, July 1999.

[2] Andreas Falkner, Gerhard Fleischanderl, Configuration requirements from railway interlocking stations, IJCAI 2001 Workshop on Configuration, Seattle, WA, Aug. 2001.

[3] Gerhard Fleischanderl, Gerhard E. Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems & their applications*, 13(4):59-68, July/Aug. 1998.

[4] Steven Minton, Mark D. Johnston, Andrew B. Philips, Philip Laird Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems, Artificial Intelligence vol.58, pp.161-205, 1992

[5] Bart Selman, Henry A. Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. *National Conference on AI* (AAAI'93), pp.46-51, Washington, D.C., July 1993.

[6] James Rumbaugh, Ivar Jacobsen and Grady Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.

Configuration Tools and Methods for Mass Customization of Mechatronical Products

Udo Pulm[1]

Abstract. This short paper presents the problems concerning modularization and configuration of functionally complex products with high requirements and a large amount of variants in mechanical engineering. By that we are searching for a software tool that focuses on the product development rather than sales and marketing or the customer. We would like to present requirements to a configuration tool or method from this point of view and where configuration may has to go. These approaches are embedded in a comprehensive research project on mass customization.

1 INTRODUCTION

Increasing market demands, both concerning higher customer requirements and pressure from competitors, force enterprises to diversify their product range. Almost each company has to deal with the problems of offering more variants and the implicit complexity of products and processes. There are many strategies and methods in mechanical engineering such as building blocks, platform strategies, modularization, standardization, complexity management, etc. The logical consequence of this growing variety seems to be mass customization, that every customer gets his own individual or at least individualized product (e. g. [1]). This strategy implicates disadvantages such as the uncertainty of customer wishes as well as advantages by shifting paradigms, i. e. to get away from classical organizations, structures, and companies.

2 MASS CUSTOMIZATION

Mass customization is the answer to the demand for more individual and diverse products. It is also the offer that each customer gets his individual product in conditions that are comparable to those of mass produced products. The product does not need to be completely individual, it just has to be individualized so far that each customer wish is fulfilled and the product fits perfectly. The degree of this individualization depends on the product and still has to be defined. Mass customization in mechanical engineering implies completely new production and logistic technologies and processes, company structures such as miniature plants near to the market, and integration of services. The very central aspect is the integration of the customer in interactive processes. Our focus is on the very different product development processes. The derivation of the individualized product has to take place in short time, e. g. one day. To achieve this, a so called product spectrum has to be developed in advance, so that the final product definition is adapted on basis of the concrete customer wishes. The product spectrum defines all the possibilities and degrees of freedom of the product and defines it as far as possible. The new aspect is that not only a specific amount of variants is offered, from which the customer can configure his products, but that also completely new components are developed, produced and integrated. The advance development mainly consists of the structure planning and the evaluation of product properties, next to the modular, parametric, or principle development of components. The difference becomes clear in a comparison with traditional design processes. Usually, there is a quite short (months) structure planning process, i. e. the conceptual design, which is mostly based on a long-term grown product architecture with only little changes. After that the actual development takes place. This design takes most of the time (years) and results in one product with a defined amount of derived variants. As last step, there is the configuration of the final product, which ideally just takes a few minutes when not regarding the decision process. The development process in mass customization consists of two phases, the development of the product spectrum, which can take months to years, and which not inevitably contains the derivation of one concrete variant (except e. g. for evaluation). This process may take longer than usual conceptual design or even than the complete development. The use of the product spectrum's potential, the adaptation to the final product definition then takes about one day, so that there is still the possibility of individualizing and developing many not predefined aspects. Another difference is that in mass customization the potential is always growing, i. e. the experiences of the single product adaptations feed back to the product spectrum and following adaptation processes.

3 EXISTING TOOLS AND METHODS

Though mass customization is a new approach, there are similarities to other methods which are recommended to be used and adapted (see [2]). The basis of existing tools and methods is the general product structure that consists of a hierarchical, "vertical" composition structure (AND-relations), and a hierarchical, "horizontal" generalization structure (OR-relations), which are both interconnected on different levels of detail. Next to this already very complex system, there are cross connections, i. e. relations such as constraints, configuration rules, or the like. At least these three relations appear and have to appear in each product model concerning configuration or variants. Most design tools based on a product structure offer these relations somehow, e. g. product data management (PDM) systems (e. g. [3]), document management systems, content management systems, ontology editors (e. g. [4]), CAE systems, configuration systems (e. g. [5]), or also bills of material. But the possibilities of these tools are not adequate to build up a respective product structure. This is the same for CAD

[1] Institute of Product Development, Technische Universitaet Muenchen, Boltzmannstr. 15, 85748 Garching, Germany, email: pulm@pe.mw.tum.de

tools, which have to be also integrated due to their significant position in product development as well as to achieve a comprehensive tool for the whole development process. A major problem may be that those systems have to be very flexibly but precise in representing the product's architecture and logical structure. Some of the consequences and problems of this are addressed and discussed in the following.

4 STRUCTURE PLANNING TOOLS

The following chapter specifies requirements as well as problems and first solutions for structure planning tools.

4.1 Requirements

Next to the above mentioned basic elements, other elements are to be represented in mass customization. These are altogether

- components, i. e. products, modules, assemblies or parts, here in the meaning of "masters", i. e. an abstract formulation (= class), e. g. body or engine of a car
- variants, i. e. a specification of a master component, e. g. 100 hp engine or 150 hp engine
- requirements, i. e. any postulated property of the product and the starting basis for the development process, e. g. cost or safety
- functions, i. e. the abstract formulation of a component's purpose in order to not being bound to a specific solution or to have a placeholder for customer wishes (possibly with input and output), e. g. transport person, transport material
- degrees of freedom, i. e. the prognosticated customer wishes and the possibilities of the product spectrum, e. g. color, size, engine
- realization possibilities, i. e. the link between the customer wishes or degrees of freedom and the concerned components, e. g. modular, parametric, completely individualized
- properties, i. e. the actual properties of the components described by criterion and specification, e. g. color red, size 5 m
- groups as a combination of different elements to support the configuration, e. g. sports gear and sports seats
- geometry, i. e. a logical decomposed description of a part on a more detailed level, e. g. surface, line, point
- all relations between the elements, as there are
  - ⇨ logical relations (is_part_of, is_a), e. g. engine is part of car
  - ⇨ configuration relations (not_with, must_with, can_with), e. g. red body not with green seats
  - ⇨ semantic relations between different elements (has), e. g. car has function transport person
  - ⇨ technical relations, i. e. different kinds interfaces between components, which can again consists of other components, e. g. interface between body and engine.

All these elements can be represented in separate hierarchical structures, but have to be connected in order to get a comprehensive product model. There may be overlaps between these elements as well as further elements such as versions, technologies, structures, customer wishes, or documents. But those elements are the substantial components of engineering design methodology. Connections are possible between any of these elements. In other words, each elements "consists" of all the other elements on a higher level of detail. Other general requirements on a software tool are the consistency of the data, stability and economy of the tool, connections to other applications, documentation of experiences, support of information flow, distributed use, etc.

4.2 General questions

We would like to address some general questions or problems concerning the modularization of a product in order to enable a configuration as well as existing tools and methods. They also reflect further requirements to such a system.

4.2.1 Emphasis of early phases instead of focus is on sales and marketing

Most configuration tools or similar products are based on an existing product structure and serve to handle or manage this structure in later phases. The "filling" of the data base is not in the centre of the regard. From our point of view, there has to be an emphasis of the early phases, where the basic product structure, which shall be oriented to variants and configuration, is defined. By that, the configuration system also becomes a tool for the engineer.

4.2.2 Representation of a product spectrum in contrast to predefined variants and flexibility of the product structure next to a predefined frame

In mass customization we do not want to just offer a specific amount of variants. Each customer wish shall be fulfilled. This means that in the structure degrees of freedom are to be represented, which may be fuzzy and cannot be totally foreseen, so that the product model has to be extensible. This is also important since the potential of the product spectrum grows with each adaptation process. It implies a certain flexibility of the product structure and the configuration tool, i. e. it should be able to represent arbitrary elements, and above all the overall structure should be easily changeable, i. e. the whole structure and not only single elements. This point represents the main demand in our approach.

4.2.3 Continuous use throughout the design process

The integration of specific elements is necessary in order to support standard procedures and techniques of design methodology (see [6]). A combination of these elements (requirements, functions, principles, drawings, etc.) has not been realized in commercial tools yet. In addition to point 4.2.1 the single phases of the product creation process have to be connected, so that there is a path from customer wishes to technical characteristics. It belongs to this point that there is not another isolated application but that the tool can be integrated into the existing applications.

4.2.4 Complexity and level of detail

One of the major problems is how to deal with complexity itself. Most methods and tools somehow help managing complexity, but how to handle systems with thousands or millions of parts and a respective amount of relations is not answered yet. Together with this there is the question for the right level of detail, i. e. how detailed the product is described. A hierarchical structure can help this problem; it also allows to combine customer views (on a low level of detail) and technical aspects (on a high level of detail). The most effective way of handling complexity seems to be systems engineering, i. e. among other things a strict systematical and modularized regard of the product. Regarding a software tool, this implies the possibility of user-specific views.

4.2.5 Modularization and standardization vs. integration

Another major problem is the contradiction between modularization und standardization. Increasing requirements, e. g. concerning safety, space, comfort, and functionalities, force engineers to design highly integrated products. This only technical solution prevents an easy modularization and by that standardization of the product. An evaluation of these contrasts is still missing.

4.2.6 Usability and implementation

The usability of the system is important for both the efficiency of the work and the acceptance of the tool. An essential aspect is the self-explanatory use. This counts for the engineer as well as for the customer. A guideline for good usability may be given by the ISO 9241. Related to this is the implementation within a company (see [7]). A tool will just be accepted if either the pressure of superiors or the benefit in the use of it is big enough. The problem with implementation is that many tools show their benefits mainly in a company wide, large scale (e. g. PDM systems), so that enormous efforts are necessary before they can be used.

4.2.7 Design process support

It is desirable that the design process itself is supported. Since design processes cannot be automated in general, the intention is on the presentation of experiences, the estimation of effects and logical networks, as well as the condensation of relevant data in order to support controlling and decisions.

4.2.8 Technical implementation

Next to the implementation of a system within a company, the technical implementation poses some questions. These are the same in an object oriented, relational, or similar approach (e. g. ontologies), and are exemplarily:

- Are properties within objects or are they objects themselves (with criterion and specification)?
- Are different object types really different objects or is there a general object with the object type as an property?
- Are there relations (v. above) between objects, are relations objects themselves, or are relations properties of objects?

These questions belong to the topic of how sophisticated, how fixed, or how flexible the system shall be, and they are very central from a pragmatic point of view.

4.3 Approaches

We have developed some tools, which now have to be adapted to the requirements of mass customization in mechanical engineering. They show approaches that help to manage the previous described problems in a first step.

4.3.1 Semantic network and systems engineering

Many tools base on a representation of the product, the respective process, and the underlying organization according to systems engineering, i. e. a both hierarchical and networked structure of elements and their relations. A previous developed tool (see [8]) offered an semantic network, where the elements could be arranged according to design methodology and where the process logic became visible. The object oriented data model could be extended in runtime. The semantic network should be represented either in a graphical network or in matrices between any two – also the same – of the above listed elements, combined with a hierarchical structure of the single elements.

4.3.2 Combination of standard applications

Another tool combined standard applications (MS Office) in order to get graphical representations as well as an underlying complex data base. This approach is also recommended in our topic since

- we need to represent the product structure graphically, store the model in an database, and have to use matrices for the relations between the objects
- the use of standard applications is easy for each user, so that the acceptance is high and there is not another isolated application, which can become obsolete. Furthermore, the existing data can be easily used in another context.

Next to a combination of these two approaches, we try to adapt existing tools to our requirements.

5 SUMMARY AND CONCLUSION

We have presented the main requirements on configuration or structure planning tools coming from mass customization in mechanical engineering. The main aspect is to manage complexity with computer support. The antagonisms of integration vs. modularization, or between flexibility and methodical support are problems and contradictions that are not solved by existing methods or applications yet. Main questions are, which application helps this situation best, how can it be used, where are its borders and how has an engineer to proceed behind these borders. The described topic is part of an project within the interdisciplinary collaborative research centre (SFB 582) "Near to the market production of individualized products", which is founded by the DFG. The problems show that the collaboration between (mechanical) engineers and software designers has to be heightened since on one side there are the problems and on the other side there are the solutions. Often enough software does not fit the problems adequately, or much efforts are spend for the search for and development of reasonably usable applications on side of mechanical engineers.

REFERENCES

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

[2] U. Lindemann and U. Pulm, *Enhanced Product Structuring and Evaluation of Product Properties for Mass Customization*. HKUST; TUM (Eds.): MCP'01, World Congress on Mass Customization and Personalization, Hong Kong, 2001. (CD-ROM)

[3] J. Schoettner, *Produktdatenmanagement in der Fertigungsindustrie, Prinzip – Konzepte – Strategien*, Hanser, Munich, 1999.

[4] N. F. Noy and M. A. Musen, *Algorithm and Tool for Automated Ontology Merging and Alignment*, Seventeenth National Conference on Artificial Intelligence (AAAI-2000), Austin, TX, 2000.

[5] A. Guenter and C. Kuehn, *Knowledge-Based Configuration - Survey and Future Trends,* F. Puppe, Expertensysteme '99, Lecture Notes, Springer, Berlin, 1999.

[6] K. Ehrlenspiel, *Integrierte Produktentwicklung*, Hanser, Muncih, 1995.

[7] R. Stetter, *Method Implementation in Integrated Product Development*, Dr. Hut, Munich, 2000.

[8] S. Ambrosy, *Methoden und Werkzeuge für die integrierte Produktentwicklung*, Shaker, Aachen, 1997.

Optimal Configuration of Logically Partitioned Computer Products

Kevin R. Plain[1]

1 CONFIGURATION GOALS

An established corporate sales tool uses Trilogy's SalesBUILDER® technology to configure a wide range of hardware and software products. The tool has evolved over time from an application capable of configuring single server systems to an application capable of configuring storage clusters involving multiple servers. In this demonstration, we examine a configuration solution for a commercially available server. This server employs a new partitioning scheme that provides hardware and software isolation. Product experts provided specifications for the configuration model. The specifications demand optimal configurations that address the characteristics of an entire system and at the same time consider the requirements of individual logical partitions. The server must be configured as a single machine from a system- wide, physical level, and each logical partition must to be handled as if it were a separate server. Each partition has its own system boards, processors, memory, operating system, software, PCI cards, and external storage. The partitions compete for shared resources such as the system board slots, I/O card cage spaces, and I/O expansion cabinets. A single server may span multiple cabinets and multiple cases within cabinets. The logical partitioning is independent from the physical configuration, but the optimization of the hardware's physical placement is greatly influenced by the logical partitioning. Features of the sales tool increase the complexity of the configurator because the tool provides users with several powerful views of the same configuration and offers incremental/interactive functionality within each view. Product experts also specified that the model should support the configuration of multiple servers that potentially share common resources such as storage cabinets.

2 CONFIGURATION VIEWS SUPPORTED

1. Diagram: The diagram view provides an abstracted representation of the physical system including information about the containment and connectivity of various components. The server's diagram needed to simplify the complex representation of partitions across multiple server systems. This was achieved by color coding the configuration. Users are able to move/add/delete/replace components via the diagram.
2. Overview: The system overview provides a hierarchical representation of the configuration with a mixture of physical and logical concepts. This view allows users to add/remove/copy partitions and to move physical components from one partition to another.
3. Worksheet: The worksheet view allows users to make specific selections about how they want the system configured. The worksheet view is directly supported by the system overview. Nodes of the system overview's tree structure can present worksheets for the selection of items within the narrow scope of the node.

[1] Trilogy, Austin, Texas, email: kevin.plain@trilogy.com

3 OPTIMIZING ON SIX DIMENSIONS

1. System Boards (cells): The position of system boards involving multiple partitions across multiple system cabinets with consideration for bus structure and bandwidth. System boards are ranked according to the nature of their partition. Partitions with more system boards are given priority.
2. Memory: Balancing across multiple system boards within a single partition. This optimization will not conflict with other optimizations.
3. Processors: Balancing across multiple system boards within a single partition. This optimization will not conflict with other optimizations.
4. I/O Boards (card cages): The position of I/O boards involving multiple partitions with consideration for the proximity of connected system boards. Optimized across multiple system cabinets and multiple expansion cabinets. A card cages may move to a less optimal position in deference to other card cages when the proximity of a connected cell presents a conflict.
5. Chassis (cases): Location within cabinets showing consideration for ergonomics, weight, and factory processes. Expansion cabinets are added as the need for them appears. If the number of card cages exceeds the number allowed in the system cabinets, the configurator will add expansion cabinets. The maximum number of system cabinets and expansion cabinets may be limited by marketing, factory, or engineering restrictions.
6. PCI Cards: The position of PCI cards within and across I/O boards of single partitions with consideration for card speed, card voltage, and I/O board designations. Some cards may be given special treatment due to the minimal requirements of the partitions. These special cards may move to a card cage connected to a specially designated cell. Explicit placement of these special cards by a user of the configuration tool may cause card cages to rearrange and reconnect to different cells.

4 COMPLEXITIES OF OPTIMIZATION WITH INCREMENTAL CONFIGURATION AND LOGICAL PARTITIONING

Optimal configuration is complicated by the incremental/interactive features of the sales tool interface, and by the interaction of optimization criteria. Users can add items in any order and at any time during the configuration process. For example, the addition of a new I/O board to a partition may cause optimization to position the new board within a system cabinet to keep it in close proximity to its connected system board. Optimal configuration may require relocation of an existing I/O board in order to provide space for the new I/O board. The seemingly trivial movement of a single PCI card demonstrates how simple user requirements can have a large impact on a configuration due to dependencies between optimized components. In this example, each partition requires a specialized PCI card and the configurator tries to keep the I/O board containing the specialized card in close proximity to the first system board of the partition. Optimization tries to balance the needs of all partitions and at the same time maintain this close proximity rule with limited I/O board spaces. The organization of the I/O boards is affected if the user drags/drops the specialized card from one location to another or adds redundant specialized cards to a partition. Interaction between optimization criteria can be seen when the user decides to add/remove system boards to/from a partition. Optimal configuration can require complete rearrangement of system boards to maximize bus characteristics. I/O boards have their own optimization characteristics with regards to system boards, and may need complete rearrangement as well. This may require some I/O boards with ideal positioning to be moved into spaces that are less ideal.

5 ADDITIONAL MODEL FEATURES

1. Upgrade scenarios
2. Save and restore of configured products in an evolving knowledge base
3. Bundling and quote of the configured product

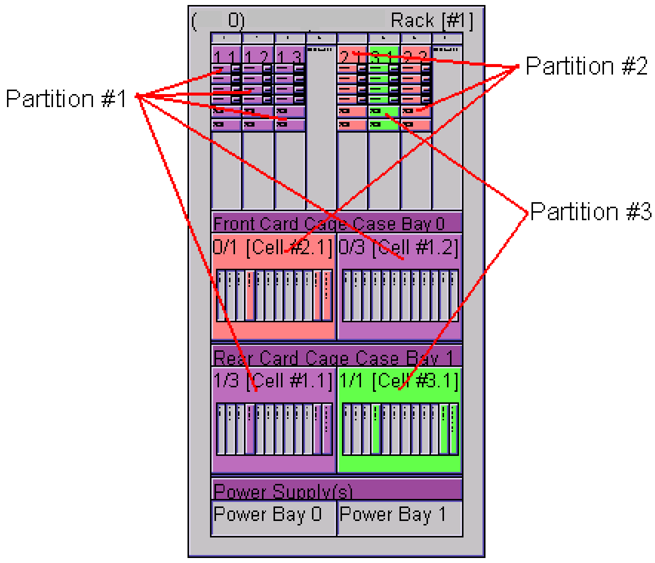


Figure 1. Logical partitioning

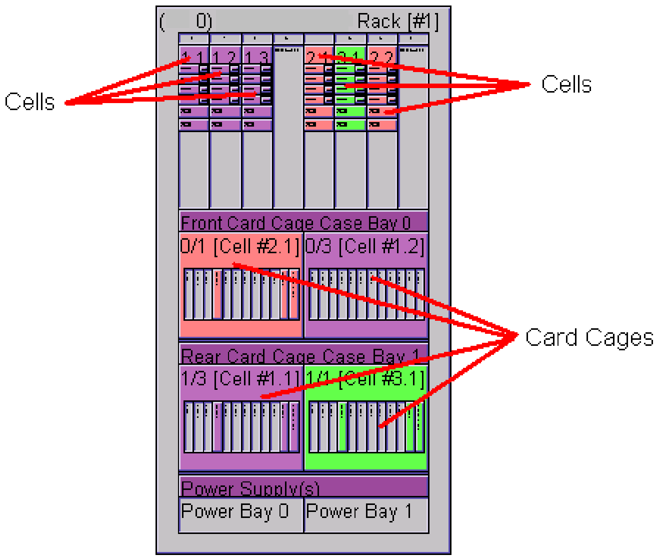


Figure 2. Reference for some of the various types of hardware that are combined to configure the server

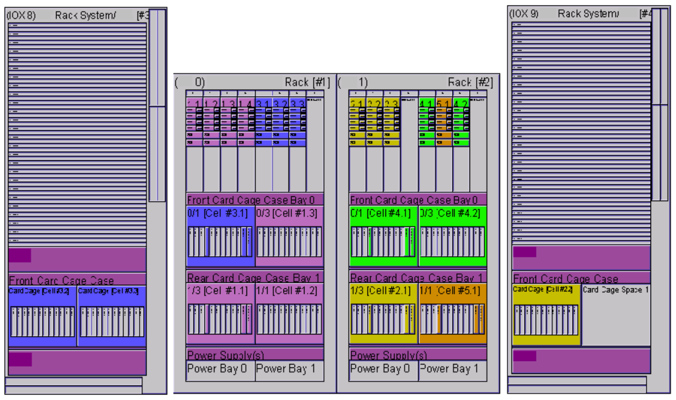


Figure 3. Example of logical partitioning and optimization across physical enclosures

Vehicle Sales Configuration : the Cluster Tree Approach

Bernard Pargamin[1]

Abstract. Most current commercial configurators, mainly based on constraint propagation, are known, among other limitations, to be unable to guarantee complete deduction in free order interactive configuration. Yet, on specific ranges of products, completeness of inference can be achieved. New approaches, based on the compilation of the system of constraints defining the diversity of the product to configure, are making their way. We present such a complete configuration engine, C2G, developed internally by Renault, with a list of basic functional requirements that we couldn't find together in any commercial configurator. The compiled approach we describe made their implementation possible, especially the much needed 'filtering on the price' and 'user driven conflict resolution'.

Basically, compilation splits the configuration variables into clusters organised in a tree. Variables in a cluster are in high interaction, while variables of different clusters are in low interaction. Inference is done locally in each cluster and the updated state of the node is propagated between nodes to provide a global solution. Inference proves to be linear in the tree structure, allowing short and predictable run-times. This representation is especially efficient for low treewidth domains, which is the case for the automotive industry.

1 INTRODUCTION

Web enabled sales configurators are widely used in the automotive industry, but the level of functionality offered by these implementations is deceptively low. A basic customer's question such as: 'I have $15,000. What is the cheapest vehicle I can buy with a diesel engine and an automatic gearbox ?' won't generate any direct answer from the online configurators of the major vehicle builders world-wide. You will have to study the documentation first, choose a model outside the configuration process, configure your vehicle with possibly inadequate cost information, eventually iterate on this process and backtrack if you are unlucky until you get what you are looking for.

Customers don't like to iterate: they want responsiveness, accuracy, full information on the consequences of their choices, free order and full freedom of choice. They want effective filtering on their budget. They don't like backtracking. If your on-line configurator can't give that, they will zap.

Renault, a major vehicle manufacturer, was bothered by this situation and articulated its configuration vision in an internal white paper (2001, unpublished). It then arrived at the conclusion that the current mainstream commercial configurators, because of their general purpose, were unable to offer this kind of rich functionalities on Renault range of vehicles, and it decided to switch to an internal configuration project: C2G (2nd Generation Configurator) in order to provide them on the web by Q3 2002.

C2G was an outcome of a 'Product Diversity Modelling' Project initiated six years ago. In this framework, Product Diversity is represented by a compiled data structure on which operates a deductively complete configuration (inference) engine whose response time is linear on the size of the compiled structure. Basically, the set of boolean variables used in the constraints restricting diversity is split into clusters by converting a graphical representation of the problem into a tree structured representation where each node in the tree represents a tightly-connected sub-problem, and the arcs represent the loose coupling between sub-problems. Inference is done locally at each node and the updated state of the node is propagated between nodes to provide a global solution. Inference proves to be linear in the tree structure.

This paper is structured as follows: We will first outline Renault's basic requirements for configuration in section 2, then we will discuss Vehicle Diversity Specification and modelling in section 3. In section 4, we will present the principles of Diversity Compilation we used in C2G, and we will show how very fast complete deductive inference can be obtained by state propagation on a cluster tree. Section 5 will focus on the problem of price representation, the only non boolean part of our data, and we will present a propagation algorithm that finds the cheapest vehicle consistent with the current state of the configuration in linear time, thus enabling fast exact filtering by a customer's constraint on the maximum price. Section 6 will outline consistency restoration in C2G, a very basic feature for a commercial configurator, and we will present some concluding remarks in section 7.

2 RENAULT'S BASIC CONFIGURATION REQUIREMENTS

The functional configuration requirements we discuss here come from our experience with the shortcomings of the mainstream commercial configurator we used previously. They reflect the views of both Renault Marketing managers and configuration specialists.

2.1 Completeness of inference should be guaranteed. This means that *all* that can be logically inferred at a given state of the process is actually inferred. No dead end, no backtracking for the user. This is absolutely the number one item on top of the requirements list. Configurator vendors say it is impossible to obtain, because of the NP-Completeness of the problem. Actually, we arrived at another conclusion, as far as automotive sales configuration is concerned.

The importance of this point is better seen if we realize that the diversity specification could be inconsistent, and the configurator wouldn't even detect it. The failure rate of the configuration would then be 100%.

[1] Renault S.A., Direction des Technologies et Systèmes d'Information 92109 Boulogne, France.
e-mail: bernard.pargamin@renault.com

2.2 Full information on the consequences of a choice should be provided (may be optionally in some cases) : excluded or implied features must be exhibited because they may cause the customer to change his choice when he sees the undesirable consequences of his projected choice. This feature is a dynamic way to inform the user of the logical constraints, at the moment they will trigger inference. Without it, the user would discover the effect of the constraints too late, and would have to force a conflicting choice, which would cause computational overhead to the configurator to restore consistency, and a real nuisance for the user. Full information includes also:

2.2.1 Pricing information: in the real life, no one makes choices without any cost consideration. You wouldn't fill your cart in a supermarket where you only discover the final price at the pay-desk. Moreover this pricing information must be accurate and guaranteed. At every step, we need the minimum price to be shown for each choice, i.e. *the price of the cheapest vehicle fulfilling the partial configuration*. This is very different to simply showing the price of an option, often irrelevant in order to show the real financial impact of a choice: if option A implies option B or option C (a very common type of constraint), in front of A you should put: Previous Min Price + price option A + min(price option B, price option C). Adding only the price of option A would be misleading.

2.2.2 Delivery timing information is also needed. This is not trivial, because you need a way to evaluate the best possible delivery date on a partially configured vehicle. This is the only way to show which specific choice makes the delay grow. An evaluation at the end of the configuration would be too late and would not say what to change in the vehicle features to improve its delivery date. In the most sophisticated implementation, this requirement is a real challenge: a centralized application must manage capacity constraints and manufacturing schedules to give accurate estimates. As a first step, an approximation based on manufacturing eligibility dates and marketing dates would probably be sufficient.

Existing stocks of readily available vehicles should be taken into account in the configuration process.

2.3 Filtering on a maximum budget and/or a maximum delivery lead time must be possible. In certain cases (fleet configuration) they are strict constraints that can't be bypassed.

2.4 Financing possibilities (loan) should be integrated in the process in the same way as the price. The user is not expected to learn at the end of a 10 minutes configuration that this vehicle exceeds its financing capability: we need a filtering capability similar to price filtering. This is much more difficult to achieve because there are several types of loan and the rate of the loan may depend on the final price of the vehicle, which is not known until the end of the configuration.

2.5 Adequate treatment of option packs must be provided. Packs generate implicit constraints that must be explicited (for instance, two packs with a common feature are exclusive, so as not to make the customer pay twice that feature). In the configuration process, they should be treated just like any other features.

2.6 Freedom of choice

- **Free configuration order**: *Configuration is a completely user driven process*: the user chooses the specification order that best fits his priorities. There is no compulsory "natural order", but of course, when the user has expressed his priorities, the configurator can choose the order that best optimizes its computational task. There is a suggested order that optimizes configuration, and if the user is pleased with it, we use it.
- **Negative choice**: A feature can be either chosen or *excluded* by the customer. Excluding a feature is a valid choice. Allowing only the selection of a feature can't prevent an undesired feature to appear inadvertently by having been implied by a constraint.
- **Permissiveness:** The user can change his mind without any restriction during the process: an inconsistent choice can be forced at any step by the customer, and it is the configuration engine's task to *restore consistency*, by proposing changes to some previous choices. This situation may arise routinely simply because the user doesn't know the logical constraints concerning features, so he may be surprised by their effect. It will happen less often if he enters his choices by decreasing priority order.

Consistency restoration must be completely *user driven*: there are usually several ways to do it, and it is the user's choice to determine which one is best for him, not the system's choice.

- **No obligation to choose**: Any choice at a given state can be delayed by the user: "I don't know yet" or "I don't bother" are perfectly acceptable answers to a proposed choice. So, the user is never stuck because he has no answer to give. The configurator will come back later on these choices, and sometimes it will not even be necessary because the answers will be automatically deduced from other user's choices. This applies of course to the model's choice, and it means that we need unrestricted transversality in the configuration process.

2.7 No prerequisite knowledge of the product should be expected from the customer. Any choice should have an associated help giving extensively the pro's and the con's of each alternative. This requirement introduces some form of *recursive configuration* inside the primary configuration. For instance the choice of a radio set is itself a second level configuration based on features such as: number of channels, power per channel, CD with charger, traffic information , …Just showing the list of the various possible radio types with their price is clearly not enough to let the user make an informed choice.

2.8 Tranversality of the configuration process is required: first generation configurators require that the process can only begin when the customer has already chosen a vehicle Model. This seems a reasonable assumption, but *it is not*. The customer is thus expected to have previously collected paper information about models and versions, studied it carefully by himself and to have made his choice alone. This means that a whole important part of the configuration process is left aside, without any assistance.

We think that Model and Version choices are obviously part of the configuration process, and that in this case, these choices might be the consequence of other features choices. Note that this reverses the traditional view that the natural order is to begin with Model/Version, and then to choose options whose availability is the consequence of the version's choice.

2.9 Automatic completion of a partial configuration must be available, with several alternate optimization criteria: cheapest, most quickly available or mixed, such as 'the cheapest vehicle under $15,000 available in less than 3 weeks'.

2.10 Assisted conflict resolution

The main configurator's task is to prevent the appearance of conflicts by automatically excluding all choices that wouldn't be consistent with the user's former choices. If the configuration engine guarantees completeness of inference, a conflict should never appear. But even in this case, the user may change his mind. He can force an inconsistent choice, and the configurator must analyze the inconsistency so as to find the minimal sets of former choices that have to be revised to restore consistency. Without this feature, the user would have to start a new configuration session and to repeat most of his previous choices.

2.11 Response time must be compatible with an interactive web usage: at most a few (< 2) seconds in the worst case.

3 RENAULT VEHICLE DIVERSITY SPECIFICATION AND MODELLING

Vehicle diversity (i.e. the number of distinct possible customer's choice, can be huge, reaching at Renault 10^{20} at the engineering stage and up to 10^{10} at the showroom. Moreover, this diversity is a source of major industrial complexity because of technical, commercial and legal constraints resulting in numerous implications and/or exclusions of features and options.

In the automotive industry, Product Diversity Specification (PDS) is addressed by an ISO norm: STEP-AP214 (ISO 10303-214:2001). In STEP terminology, vehicle descriptive features or attributes are known as *specifications* (abbrev. *specs*), grouped in *specifications_categories* (abbrev. *spec_cat*).

A spec_cat is a variable whose discrete possible values are the specs. For example we have a spec_cat 'gearbox type' with 2 specs: 'manual' and 'automatic'. A spec is an instantiation of a spec_cat, and we will attach a boolean variable (abbrev. *bvar*) to each spec, among which we will find our configuration variables.
The set of spec_cat, called *lexicon*, is globally common to all models of the constructor's range, with some exceptions. A specific vehicle is uniquely defined by a tuple of specs, one and only one for each spec_cat of the lexicon.

A subset of the vehicle range is intentionally defined by a boolean formula on specs, called a *specification_expression* (abbrev. *spec_expr*).

Vehicle diversity is usually not practically enumerable, and this prevents an extensional definition such as a simple enumeration. Diversity must be intentionally specified, by spec_expr.

The basic idea is that the entire combination of specs is allowed, except those that are explicitly excluded by boolean constraints expressed by spec_expr. The task of PDS is to build, control, store and retrieve these constraints.

Actually, for ergonomic and convenience reasons, engineering staff in charge of PDS do not usually directly manipulate spec_expr, but instead they complete compatibility tables of various shapes and size, they explicitly enumerate valid tuples of specs on selected subset of the lexicon, and sometimes they write spec_expr in a simplified syntax.

The main point is that PDS of any form can be translated into boolean constraints in linear time, giving a CDNF (Conjunction of Disjunctive Normal Form). Alternatively, a CSP (Constraint Satisfaction Problem) formalism can be used, but it is known to be equivalent.

Implicit constraints due to the spec_cat/spec structure must be explicitly added: inside a spec_cat, any 2 specs are exclusive and the disjunction of the specs is TRUE.

Our model of vehicle diversity is thus a pure boolean CDNF, equivalent to a propositional theory, and in this first step, we have transformed Renault PDS into a propositional knowledge base (PKB) able to feed a configuration engine. In this model, most interesting tasks become *reasoning* tasks, involving logical inference [21, 19, 20] , such as satisfiability tests and clausal entailment.

This first modelling step is necessary but is not sufficient, because as it turns out, most available complete algorithms for those tasks are exponential in the number of variables or in the number of constraints. Brute force, even with gigahertz computers, leads us instantly to exponential blow-up.

A typical Renault Model is described by a lexicon of 100 spec_cat containing about 400 specs. The CDNF has about 2 to 3000 conjunctive terms and commercial diversity (size of the search space for the configurator) ranges from 10^3 to 10^{10} . At the engineering stage, diversity is much higher, reaching 10^{20}.

An example of translated PDS for a Renault X64 model can be found for benchmarking purpose at :
http://www.irit.fr/ACTIVITES/RPDMP/CSPconfig.html
A cnf sample file is also available on demand (10813 clauses on 658 variables)

4 PRINCIPLES OF RENAULT CONFIGURATION ENGINE C2G

We have to deal with the well known NP-completeness of the propositional satisfiability problem, and consequently of boolean configuration. But NP-completeness doesn't prevent polynomial or even linear solutions in specific domains. While a general polynomial time configurator is impossible, a linear time vehicle configurator proves possible if you can exploit the structure of the domain.

C2G is fully based on the standard compiled representation of Renault vehicle diversity in the form of a cluster tree that has been used in various applications since 1995 . Compiled approaches for propositional theories, increasingly popular, consist in investing once in a heavy off-line compilation whose computational overhead can be amortised with many fast on-line queries. The size of the compiled structure is not directly related to the number of models of the theory and can be exponentially smaller. BDD (binary decision diagrams [5]), finite automata (a variant of BDD [9, 18]), DNNF (Deterministic negation normal form [7]), prime implicates [15] are mainly used to this effect. See also [13] for a classification of compiled approaches. We must stress that there is no *best* compilation, because it depends on the structure of your problem and on what you intend to do with your data. We, at Renault, decided to use cluster tree compilation, as the most effective and efficient representation of vehicle diversity, seen as a propositional theory, for deductive and probabilistic inference.

Cluster trees are well known in the field of probabilistic inference, especially in the Bayesian Network area. A main advance was given by Lauritzen and Spiegelhalter [6, 11] with the use of a secondary structure, the cluster tree (also called join tree, junction tree, clique tree) and an exact probabilistic inference algorithm based on Probability Propagation in the Cluster Tree (PPCT). See [10] for a procedural description of PPCT. We are not aware of any significant use of cluster trees in deterministic inference problems, except by Darwiche [7, 8] and Dechter [22].

4.1 Construction of the cluster tree

A cluster tree is a tree whose nodes are clusters of variables, satisfying the running-intersection property: if a variable is shared by 2 nodes, it must be present in every node on the path between the 2 nodes. This property is needed for the completeness of inference algorithms operating on the cluster tree. The *treewidth* of the cluster tree is the number of variables in the largest cluster. The best cluster tree is the one with minimal treewidth w* (the size of the cluster tree is exponential in w*). In a sense, the treewidth is a measure of the connectivity of the constraints graph.

We are looking for the minimal treewidth cluster tree in which every constraint fits in at least one cluster (i.e. all the bvars of the constraint belong to this cluster).

We first build the *interaction graph* of the boolean constraints in the following way: we create a node for each boolean variable, and put an edge between node i and j if v_i and v_j appear simultaneously into a constraint. This graph shows graphically the structure of the problem.

We then built the cluster tree with the smallest possible treewidth. This is a NP-hard problem, but a number of algorithms are available in the literature for computing a close to optimal cluster tree from an acyclic undirected graph. [2, 3]. Moreover, a linear time algorithm for finding tree decompositions of small treewidth was presented by Bodlaender [4]

4.2 Cluster implementation

Clusters are supposed to be small enough so that brute force can solve any inference problem inside the cluster. Yet, a clever implementation is still of utmost importance to improve speed and allow for bigger clusters. First we build the set of all boolean variables instantiations consistent with the constraints that fit in the cluster (logical models). We associate to each bvar x_i a boolean vector (VBV) whose ith position is set to TRUE if x_i is TRUE in the ith model, FALSE otherwise.

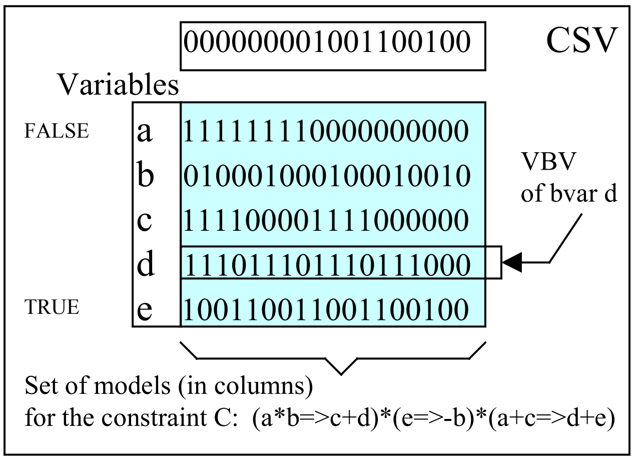


Figure 1. A cluster implementation

We then build a cluster state vector (CSV) a boolean vector whose ith position is set to TRUE if the ith model is consistent with the current state of knowledge on the values of the bvar of the cluster, FALSE otherwise.

During the configuration process, when a bvar is instantiated by the user, the CSV is re-evaluated by a boolean AND operation between the prior CSV and the VBV or its negation. The effect of the change is then propagated to all other bvar of the cluster that are deduced to TRUE (resp. FALSE) if its VBV (resp. the negation of its VBV) is compatible with the CSV.

Local inference in the cluster is sound and complete in linear time in the number of models. In the worst case, this size is exponential in the number of variables, but as a cluster groups a small number of variables with high interaction, the real size is usually much lower. This cluster implementation has two nice properties:

- the more we add constraints to the cluster, the smaller its size is.
- All operations are vector boolean operations, taking advantage of internal register parallelism on 32 or 64 bit words.

4.3 Propagation on the cluster tree

We choose an arbitrary cluster as the root of the tree.

Two adjacent nodes of the cluster tree share variables. We can treat these variables as a cluster, named the *sepset*, that we introduce on the graph as a new node between the two clusters.

Propagation of state from cluster i to parent cluster j involves 2 phases:

- *Marginalisation*: propagation from cluster i to sepset ij: we set to FALSE all positions of the CSV of the sepset whose compatible positions in the CSV of cluster i are all FALSE.
- *Dispatching*: propagation from sepset ij to cluster j: For every position k of the sepset with a FALSE value, we set to FALSE all compatible positions of the CSV of the cluster j.

Whenever the state of a cluster changes, we make a global propagation on the whole structure, by propagating the change up to the root following the path of the cluster tree, and then propagating down to the leaves. During this process, whenever a cluster's state changes, it propagates the change to all the variables in the cluster.

In this way all bvar truth values that can be deduced from a state change are deduced, and we have a deductively complete truth maintenance system.

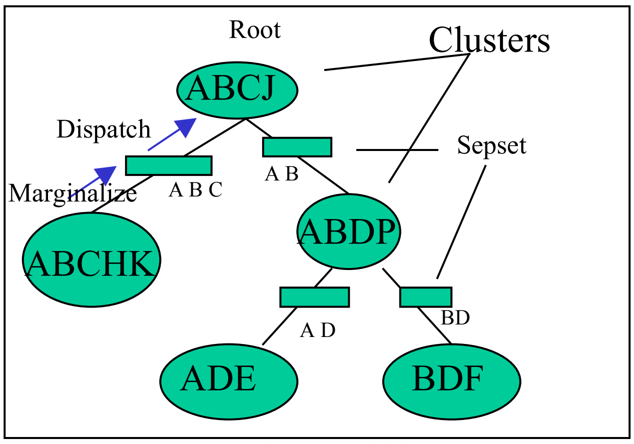


Figure 2. A cluster tree with its sepsets

4.4 Main benefits of the cluster tree

Cluster tree compilation is efficient and compact because logical independence is built in the structure: The compiled structure fully exploits the factorisation resulting from the logical conditional independence relations expressed graphically by the cluster tree: any two nodes and their descendants are logically independent given an instantiation of their shared variables.

4.5 Characteristics of Renault's cluster trees

The cluster tree structure is not related in any way to the physical structure of the car. The treewidth is around 40 to 50 for our model's cluster tree representation, depending on the model. But, as noted above, the constraints application strongly reduces the size of the biggest cluster, usually around a few hundreds columns.

Constraints can change every week, so a full offline compilation has to be done 4 to 6 times a month for each country: 10 to 15 cluster trees are compiled in about 2 minutes, with a total storage requirement of 300 to 400 KB for the logical structures.

4.6 Other uses of the cluster tree

The cluster tree is a highly tractable compiled representation of diversity for Renault vehicles: a number of different NP-complete real life problems involving diversity can be solved with it in polynomial time, and often in linear time.

- Probabilistic applications: Probabilistic constraints used to express vehicle forecast fit in the cluster tree, and by solving a linear problem, we can compute a factorised joint probability distribution (JPD) consistent with the forecast. The global JPD is split into local JPD, expressed at the cluster level by a probability vector for each model of the cluster.
- Propagation algorithms: the same scheme of propagation up and down allows for rapid calculus of the
    - cardinality of the solutions space
    - maximum entropy distribution by iterating propagations
    - entropy of a factorised jpd, cross entropy between 2 JPD's
    - probability of a variable under evidence on other variables

Within Renault Information Systems, they are used routinely in various applications, with about 60,000 queries per day:

- Consistency of Product Diversity Specification
- Consistency of parts documentation with diversity specification
- Consistency of vehicle forecast with diversity specification
- Vehicle forecasting with Maximum Entropy completion of incomplete forecast
- Engineering and Commercial vehicle configuration

5 FILTERING ON A MAXIMUM PRICE

In theory, we could handle the price in a pure boolean way:
We add a new spec_cat 'Price' with specs ranging from \$5,000 to, say, \$50,000 (by increments of \$1), and calculate the price of every vehicle configuration, then, for every value x we build the boolean constraints expressing the equivalence between spec \$x and the DNF of all vehicle configurations costing \$x.

Practically, it would be intractable, and we will of course proceed in another way: The idea is to represent the cost function as a utility function that can be factorised on the cluster tree, and to use a propagation algorithm that finds the cheapest vehicle compatible with the current state of the configuration in linear time, thus enabling fast exact filtering by a customer's constraint on a maximum price.

5.1 Price List external specification

The total cost of a vehicle is the sum of pricing elements that apply only for certain vehicles. There is a price for a version, and prices for the various options available on this version.

A Price List is a list of *pricing elements*, each consisting of a set of 2 elements: a spec_expr and a price.

5.2 Price List internal representation

We assume that every spec_expr fits in at least one cluster. If it were not the case, we would rebuild the cluster tree to satisfy this condition. We associate a numerical Price Vector to the (boolean) state vector of each node and sepset of the cluster tree.

The ith position is set to the sum of all the prices associated with spec_expr satisfied with the partial instantiation i of specs in the cluster. In this way, the price vector stores the contribution of each partial instantiation of the bvars of the cluster to the global price.
Propagation of price vector from cluster i to parent cluster j involves 2 phases:

- *Marginalisation*: propagation from cluster i to sepset ij: set every position k of the price vector of sepset ij to the min of its compatible positions in the price vector of cluster i.
- *Dispatching*: propagation from sepset ij to cluster j: For every position k of the sepset, set all compatible positions of the price vector of the cluster j to the kth value of the price vector of the sepset.

We are now ready to describe our algorithm MPP (Minimal price by Propagation):

5.3 Algorithm MPP

```
1   Initialise each cluster's Price Vector
2   For each node, from the leaves up to the root:
      -  Combine the direct descendant messages by adding their
         propagated price vector
      -  Propagate up:
           -  Marginalize min price on the sepset
           -  Dispatch up
3   Select the min price on the price vector of the root
```

6 RESTORING CONSISTENCY

This is a major feature for a commercial configurator, as it has been discussed in a earlier section.

Whenever a contradiction is deliberately entered by the user, C2G finds a minimal subset of inconsistent former choices, in linear time in the number of former choices. The user then has to change at least one of these choices. This change may in turn create a new contradiction, so the process is recursive. The maximum price given by the user is treated like any other vehicle feature's choice. Very commonly, there will be an inconsistency between the maximum price and the set of n options chosen, showing a n+1 choices inconsistency, that can be eliminated by changing the maximum price or dropping any subset of options, at the user's request. In any case, restoring consistency is a fully user-driven process.

Finding a minimal subset of inconsistent former choices is performed in this way:
We maintain an ordered list of user choices. The completeness of inference of G2G guarantees that the contradiction is detected when it happens, on the last choice. We then reorder the list by putting this last choice in first position, we make a full roll-back in the configuration session, and a roll-forward with the reordered list, reintroducing one by one each former choice. We will necessarily encounter a new contradiction, that will give us the 2nd term of the contradiction, and so on until all terms of the contradiction are grouped in the beginning of the list.

7 SUMMARY AND CONCLUSION

Modelling vehicle diversity by a system of boolean constraints is a first necessary step, but it is insufficient to solve efficiently NP-Complete problems of interest such as clausal entailment, and especially to feed a complete configurator.

Renault C2G configurator does not make inferences by constraint propagation, but adds a second crucial step by compiling the constraints into a secondary structure, a cluster tree on which complete deductive inference by state propagation is linear on the size of the compiled structure (which in turn happens to be exponential in the treewidth w* of the interaction graph). Provided that w* is small, we can perform complete inference with a bounded response time guarantee.

As it happens, small treewidth is the general case for vehicles diversity, as well as many other industrial domains, and our approach has a certain generality.

The key step to transforming a NP-Complete general problem into a linear time algorithm is thus to exploit the *structure* of the problem, as expressed by the connectivity of the interaction graph. The small treewidth of this graph allows for fast dynamic construction of the cluster tree from the boolean constraints and for compact representation of the nodes of the cluster tree.

This off-line compilation allows for very fast predictable on-line queries, and the computational overhead of the compilation can be easily amortised with thousands of queries. Compilation pays for itself.

Propagation algorithms we developed are closely related to probability propagation (see [6, 10, 11, 15]).

These ideas were implemented at Renault in a new configuration engine, C2G, that will be used by Q3 2002 for commercial configuration applications, proving their robustness in real life industrial-strength problems.

REFERENCES

[1] Amir, E and McIlraith, S (2001) *Theorem proving with structured theories,* 17th Intl' Joint Conference on Artificial Intelligence (IJCAI'01), 2001 http://www-formal.stanford.edu/eyal/papers/oor-theory-1.0-ijcai-final.ps

[2] Amir, E (2001) *Efficient Approximation for Triangulation of Minimum Treewidth,* 17th Conference on Uncertainty in Artificial Intelligence (UAI '01), 2001. http://www-formal.stanford.edu/eyal/papers/decomp-uai01-final-1.0.ps

[3] Becker, A and Geiger, D. *A Sufficiently Fast Algorithm for Finding Close to Optimal Junction Trees* In Proceedings of the 12th Annual Conference on Uncertainty in Artificial Intelligence 1996

[4] Bodlaender, H. L. (1992*) A linear time algorithm for finding tree-decompositions of small treewidth.* ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1992/1992-27.pdf

[5] Bryant, R. (1992) *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams.* In ACM Computing Surveys Vol. 24 n° 3. http://www.cs.cmu.edu/~bryant/pubdir/acmcs92.ps

[6] Cowell, Dawid, Lauritzen, Spiegelhalter. (1999) *Probabilistic Networks and Expert Systems*, Springer Verlag .

[7] Darwiche, A. (1999) *Compiling knowledge into decomposable negation normal form.* In Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), 1999 http://singapore.cs.ucla.edu/darwiche/dnnf.ps

[8] Darwiche, A. (2000) *On the Tractable Counting of Theory Models and its Applications to Belief Revision and Truth Maintenance.* in International Workshop on Belief Change & Journal of Applied Non-Classical Logics, 2000 http://www.cs.ucla.edu/~darwiche/count.ps

[9] Fargier, H and Amilhastre, J (2000) *Handling interactivity in a constraint based approach of configuration* ECAI 2000

[10] Huang C. , Darwiche A. (1996) *Inference in belief networks: A procedural guide.* International Journal of Approximate Reasoning, 15(3):225-263. http://www.cs.ucla.edu/~darwiche/ijar95.pdf

[11] Lauritzen, S. L. and Spiegelhalter, D. J.(1988) *Local computations with probabilities on graphical structures and their application to experts systems. In journal of the Royal Statistics Society,* B, 50,157-224 1988

[12] Mathieu, P and Delahaye, J.P. (1994) *A kind of logical compilation for knowledge bases.* Theoretical Computer Science 131 (1994) 197-218 ftp://ftp.lifl.fr/pub/projects/achievement/tcs94.ps.gz

[13] Marquis, P and Darwiche, A (2000), *A Perspective in Knowledge Compilation* In IJCAI-01. http://www.cs.ucla.edu/~darwiche/ijcai-01.ps

[14] Marquis, P (1995) *Knowledge Compilation Using Theory Prime Implicates*. IJCAI 1995

[15] Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* Morgan Kaufmann, San Mateo, California, 1988.

[16] Roussel, O (1997) *L'achèvement des bases de connaissance en calcul propositionnel et en calcul des prédicats.* Thèse de doctorat. Lille 1997 ftp://ftp.lifl.fr/pub/projects/achievement/these_roussel.ps.gz

[17] Schrag, R and Miranker, D. (1996) *Compilation for critically constrained Knowledge Bases.* submitted to the Journal of Artificial Intelligence Research. http://www.iet.com/users/schrag/my-papers.html

[18] Veron, M Fargier, H and Aldanondo, M (1999) *From CSP to configuration problems* AAAI-99 Workshop on Configuration http://wwwold.ifi.uni-klu.ac.at/~alf/aaai99/08Veron.doc

[19] C. Sinz, A. Kaiser, W. Küchlin (2000) *SAT-Based Consistency Checking of Automotive Electronic Product Data.* Presented at the ECAI 2000 Configuration Workshop, Berlin. http://www-sr.informatik.uni-tuebingen.de/projects/pdm/ecai2000.ps

[20] C. Sinz, A. Kaiser, W. Küchlin (2001) *Detection of Inconsistencies in Complex Product Configuration Data Using Extended Propositional SAT-Checking.* Proceedings of the 14th International FLAIRS Conference, AAAI Press

[21] Kaiser, A. and Küchlin, W (2001) *Automotive Product Documentation.* Proceedings of the 14th International IEA/AIE Conference, Springer, 2001.

[22] Dechter, R and Pearl, J (1989) *Tree Clustering for Constraint Networks*, Artificial Intelligence pp. 353-356 1989 http://www.ics.uci.edu/~csp/r06.pdf

Constraint-based Product Structuring for Configuration

Barry O'Sullivan[1]

Abstract. Traditionally, applications of Artificial Intelligence have regarded configuration as the problem of arranging parts, from predefined sets of alternatives, in a manner consistent with a predefined product structure. While such work is valuable, and represents a significant portion of the AI body of research, real-world configuration is a more complex task. In real-world configuration, the development and maintenance of the initial product structure is a core activity. Alternative product structures provide a basis for developing product families and variants, desirable for satisfying market demands for customized product delivery. In addition, the availability of product structuring rationale can be exploited when attempting to support product re-configuration in the field and recognizing redundancies in product configurations. This paper presents a constraint-based approach to supporting the synthesis of alternative product structures for configuration. The approach is based upon an expressive and general technique for modeling: the design knowledge which a designer can exploit during a design project; the life-cycle environment which the final product faces; the design specification which defines the set of requirements that the product must satisfy; and the alternative structures that are developed by the designer.

1 INTRODUCTION

Configuration is becoming a well studied design activity. In particular, there has been a growing interest in issues such as diagnosis of knowledge-bases for configuration [4], explanation generation [5] and tradeoff generation for interactive configuration [6].

While such work is valuable, and represents a significant portion of the AI body of research, real-world configuration is a more complex task. In real-world configuration, the development and maintenance of the initial product structure is a core activity and warrants further study. Alternative product structures provide a basis for developing product families and variants, desirable for satisfying market demands for customized product delivery. In addition, the availability of product structuring rationale can be exploited when attempting to support product re-configuration in the field and recognizing redundancies in product configurations.

This paper presents a constraint-based approach to supporting the synthesis of product structures for configuration. This synthesis process can be regarded as an instance of conceptual design [7]. The approach is based upon an expressive and general technique for modeling: the design knowledge which a designer can exploit during a design project; the life-cycle environment which the final product faces; the design specification which defines the set of requirements that the product must satisfy; and the alternatives structures that are developed by the designer. The approach also facilitates the development, evaluation and comparison of alternative product structures and product configurations, but due to space reasons this issue is not discussed here (see [7] for a detailed discussion).

The remainder of the paper is organized as follows. Section 2 presents a brief overview of the theory of conceptual design for configuration upon which the research presented in this paper is based. Section 3 discusses how this theory can be modeled in a constraint programming language. Section 4 presents a simple case-study which highlights some pertinent details of approach presented here. In Section 5 a number of concluding remarks are made.

2 A THEORY OF CONCEPTUAL DESIGN

The model of conceptual design adopted in this research is based on the hypothesis that during the conceptual design process a designer works from an informal statement of the requirements that the product must satisfy and generates alternative configurations of physical elements which satisfy them. These physical elements, referred to in this research as "*design entities*", can be regarded as proto-components, between which are defined a set of context relationships restricting the nature of the interfaces between the design entities. During a configuration session with a human user, these proto-components are extended into parts. In addition, as parts are being introduced into the scheme the precise meaning of the appropriate context relationships becomes known and define the interfaces between them.

Central to the process of product structuring is an understanding of function and how it can be provided. The process involves the development of a function decomposition which provides the basis for a configuration of physical elements that form a scheme. The configuration of physical elements with partially specified interface definitions between them can be regarded as a scheme for a set of *product structures*. Depending on the degree of detail that the designer has specified, each product structure will either be very general or specific. Since, in the configuration domain, we are generally only concerned with problems where the sets of parts and interfaces are known. The product structuring problem is concerned with developing a sufficiently detailed scheme from which the customer can configure the product of choice.

In the remainder of this section a brief overview of the theory of conceptual design of product structures used in this research will be presented. It will be shown how the approach presented here can support design for configurability by supporting the synthesis of product structures which contain all of the necessary constraints required to define a particular configuration problem. For a more complete discussion of the theory the reader is encouraged to refer to the more detailed literature available [7].

2.1 The Design Specification

The conceptual design process is initiated by the recognition of a need or customer requirement. This need is analyzed and translated

[1] Cork Constraint Computation Centre, Department of Computer Science, University College Cork, Ireland – (`b.osullivan@cs.ucc.ie`)

into a statement which defines the function that the product should provide (referred to as a *functional* requirement) and the *physical* requirements that the product must satisfy. This statement is known as a *design specification*.

A design specification will always contain a single functional requirement, since this represents the highest level of abstraction which defines the purpose of a product.

In addition, two classes of physical requirement can be identified: product requirements and life-cycle requirements. A product requirement can be either a *categorical requirement* that defines a relationship between attributes of the product or it can be a *preference* related to some subset of these attributes. A life-cycle requirement can be either a *categorical requirement* that defines a relationship between attributes of the product and its life-cycle, or it can be a *preference* related to some subset of these attributes.

2.2 Conceptual Design Knowledge

During conceptual design, the designer must synthesize a product structure defined in terms of physical elements which satisfies each of the functional and physical requirements in the design specification. To do so, the designer needs considerable knowledge of how function can be provided by physical means. Often, this knowledge exists in a variety of forms; a designer may not only know of particular parts and technologies that can provide particular functionality, but may be aware of abstract concepts which could also be used. For example, a designer may know that an electric light-bulb can generate heat or, alternatively, that heat can be generated by rubbing two surfaces together. The latter concept is more abstract that the former. In order to effectively support the human designer during conceptual design, these alternative types of design knowledge need to be defined and modeled in a formal way.

2.2.1 The Function-Means Map

The notion of the *function-means tree* has been proposed by researchers from the design science community as an approach to cataloging how function can be provided by means [1]. The use of function-means trees in supporting conceptual design has attracted considerable attention from a number of researchers [3].

Here a generalization of the function-means tree, called a *function-means map*, is used to model functional design knowledge [7]. In a function-means map two different types of means can be identified: a *design principle* or a *design entity*.

A design principle is a means which is defined in terms of functions that must be embodied in a design in order to provide some higher-level functionality. Design principles are abstractions of known approaches to providing function. By utilizing a design principle during product structuring, the designer can decompose higher-level functions without committing to a physical solution too early in the design process. The functions that are required by a particular design principle collectively replace the function being embodied by the principle. The functions which define a design principle will, generally, have a number of *context relations* defined between them. These context relations describe how the parts in the scheme, which provide these functions, should be configured in abstract terms so that the design principle is used in a valid way. An example design principle based on the abstraction of a bicycle is show in Figure 1. In this figure functions are illustrated as round-edged boxes, context relations are represented in dashed lines. Note that the design principle is not just a model of a known physical design solution, but is an

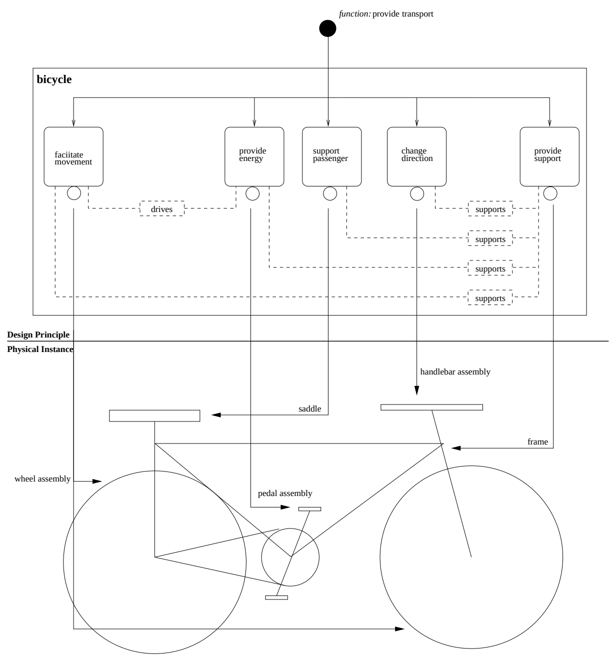


Figure 1. Abstracting an example design principle from a bicycle.

abstraction which can be used to encourage creativity and analogical reasoning during design.

A design entity, on the other hand, is a physical, tangible means for providing function. A design entity is defined by a set of parameters and the relationships that exist between these parameters. For example, an electronic resistor would be modeled as a design entity which is defined by three parameters, resistance, voltage and current, between which Ohm's Law would hold.

2.2.2 Embodiment of Function

As the designer develops a scheme for a product structure every function in the scheme is embodied by a means. Each means that is available to the designer has an associated *set of behaviors*. Each *behavior* is defined as a set of functions that the means can be used to provide simultaneously. Each behavior associated with a design principle will contain only one function to reflect the fact that it is used to decompose a single function. However, a behavior associated with a design entity may contain many functions to reflect the fact the there are many combinations of functions that the entity can provide at the same time. For example, the bulb design entity mentioned above may be able to fulfill the functions *provide light* and *generate heat* simultaneously. However, when a design entity is incorporated into a scheme (for the purpose of supporting functionality provided by one of its behaviors), it is not necessary that every function in this behavior be used in the scheme. In this respect redundant functionality may be introduced into a scheme which may become useful at a later stage when trying to incorporate new functionality into the scheme during a re-configuration exercise.

2.3 Scheme Configuration using Interfaces

When a designer begins to configure a scheme for a product structure, the process is initiated by the need to provide some functionality. The designer considers the various means that she has available in her design knowledge-base. Generally, the first means that a designer will select will be a design principle. This design principle will substitute the required (parent) functionality with a set of child functions. Ultimately the designer will embody all leaf-node functions in the scheme with design entities. During this embodiment process the context relations, from the design principles used in the scheme, will be used as a basis for defining the interfaces between the design entities used in the scheme. The precise nature of these interfaces cannot be known with certainty until the designer embodies functions with design entities; this is because the link between functions and design entities is generally not known with certainty during the development of the function decomposition for the scheme. Once the set of design entities, and the interfaces between them, are known the resultant scheme can be regarded as a product structure which defines the configuration problem. Until the precise nature of a particular interface is known, they are modeled as relations between design entities which can be used to reason about the product structure; for example, interfaces may represent simple spatial relationships which can inform an evaluation related to the relative position of parts in a product. In Section 3.2.4 a detailed example will be discussed.

The types of interfaces that may be used to synthesis a product structure will be specific to the engineering domain within which the designer is working. Indeed, these interfaces may also be specific to the particular company to which the designer belongs in order to ensure the configurability of the product.

3 CONSTRAINT-BASED IMPLEMENTATION

The implementation language used in this research was an extended form of Galileo [2], a frame-based constraint programming language based on the First-Order Predicate Calculus.

It is believed that the approach to product structuring presented here has wide applicability; it has been already tested in a variety of real-world domains. Consequently, its implementation can distinguish between those features which are generic to all applications and those features which are specific to individual design/configuration domains. In the following sections important aspects of the details of the constraint-based implementation of the design theory presented in Section 2 will be presented. For a more detailed treatment, the reader is encouraged to refer to the literature [7].

3.1 Implementing Generic Concepts

In Figure 2 the Galileo model of a generic scheme is presented.

```
domain scheme
    =::= ( scheme_name : string,
           structure   : embodiment ).
```

Figure 2. The representation of a generic scheme.

Since a scheme exists solely to provide the functionality required in the design specification, its structure should be the embodiment of that functionality. This model is based of the fact that the designer is mostly concerned with producing *embodiments* for *intended functions* by *choosing*, from among the *known means*, those which will provide the required functionality (Figure 3).

```
domain embodiment
    =::= ( hidden scheme_name : string,
           intended_function  : func,
           chosen_means       : known_means,
           reasons            : set of func_id ).
```

Figure 3. Modeling the embodiment of a function.

The same type of functionality is frequently needed in different parts of a scheme. Thus, we must represent, not a function, but an *instance* of a function. The definition of an instances of a function, referred to here as a `func`, is presented in Figure 4.

```
domain func
    =::= ( verb : string,
           noun : string,
           id   : func_id ).
```

Figure 4. Modeling a function instance in Galileo.

The final field in the definition of an `embodiment` (Figure 3) is called `reasons`. The `reasons` field associated with an `embodiment` records the motivation for the existence of the `embodiment`. It does so by recording the identity numbers of the function instances whose provision introduced the need for the embodiment. The `reasons` field of an `embodiment` provides the basis for identifying those design entities between which context relations must be considered.

3.1.1 A Generic Model of Means

Figure 5 illustrates how the generic notion of a means can be modeled. Note that there are two kinds of means: principles and entities. The final field in the definition of the generic notion of a `means`, called `funcs_provided`, is used to store which function instances within a scheme the `means` is being used to provide. Of course, a `means` should be used to provide only those function instances which it is capable of providing; this requirement is captured by a universally quantified constraint. The definition of the relation `is_a_possible_behaviour_of` is an application-specific concept.

```
domain means
    =::= ( hidden scheme_name : string,
           type               : means_type,
           funcs_provided     : set of func_id ).

domain means_type
    =::= { a_principle, an_entity }.

all means(M):
    is_a_possible_behaviour_of( M.funcs_provided, M ).
```

Figure 5. Modeling a design means in Galileo.

Based on the generic notion of a means, generic definitions for design principles and design entities can be defined. The generic notion of a `principle` and an `entity` are defined in Figure 6 as specializations of the generic notion of a `means`.

```
domain principle
    =::= { P: means(P) and
              P.type = a_principle }.

domain entity
    =::= { E: means(E) and
           E.type = an_entity }.
```

Figure 6. Generic design principle and design entity models.

3.1.2 Context Relationships and Entity Interfaces

Eventually, each embodiment is realized by the introduction of design entity instances. Thus, to ensure that design entities are configured appropriately, the context relationships between embodiments, due to any design principles used during the function decomposition, will have to be realized by interfacing appropriately the entity instances which realize the embodiments. The generic definition of an interface is is provided in Figure 7.

```
domain interface
    =::= ( hidden scheme_name : string,
           entity_1 : entity_id,
           entity_2 : entity_id ).

all interface(I):
    exists entity(E1), entity(E2):
        I.entity_1 = E1.id and
        I.entity_2 = E2.id and
        is_in_the_same_scheme_as( I, E1 ) and
        is_in_the_same_scheme_as( I, E2 ).
```

Figure 7. Modeling generic interfaces between design entities.

It can be seen that an `interface` is defined between a pair of entities. We shall see later how this generic definition of an interface can be used to define application-specific interfaces for embodying context-relations.

3.2 Implementing Application-Specific Concepts

The generic concepts introduced above serve as the basis for defining the application-specific concepts that are needed to support product structuring. The definition of these application-specific concepts in terms of the generic concepts will now be considered briefly.

3.2.1 Defining Known Means

The various design principles and design entities that are approved for use by designers working for the company constitute a set of `known_means`. The functionality that is offered by these `known_means` must be known in advance. In addition, each of these principles and entities must be described in detail by defining them as specializations of the generic representation of a means.

```
domain known_means
    =::= { an_axle, a_bicycle, a_wheel_assembly, ... }.

relation can_simultaneously_provide( known_means,
                                     set of func )
    =::= { ... }.
```

Figure 8. Defining of known means available to designers.

In Figure 8 an example of a small collection of means which are available to a designer working for Raleigh Leisure Vehicles Limited is presented. As well as listing the `known_means` that are available to designers working for a company, the company knowledge-base must specify the functions that each `known_means` can provide. This is done by declaring the relation called `can_simultaneously_provide`. This relation describes the functionality that can be simultaneously provided by the various `known_means` available to engineers working for a particular company.

3.2.2 Defining Company-specific Design Principles

Application-specific design principles can be defined as specializations of the generic design principle (Figure 6). Consider, for example, a principle based on a bicycle is defined in Figure 9.

This application-specific principle is defined to be a specialization of the generic notion of a principle. It was seen in Figure 1 that a `bicycle` principle involves five `embodiments`. The context relationships between the embodiments which are shown in Figure 1 are also defined. For example, a `drives` relationship must exist between the `embodiment e2` and `embodiment e1`. These context relationships constrain the manner in which the design entities in the final product structure are configured.

```
domain bicycle
    =::= { B: principle(B) and
           exists( B.e1 : embodiment ) and
           exists( B.e2 : embodiment ) and
           exists( B.e3 : embodiment ) and
           exists( B.e4 : embodiment ) and
           exists( B.e5 : embodiment ) and
           provides_the_function( B.e1.intended_function,
                         'facilitate', 'movement' ) and
           provides_the_function( B.e2.intended_function,
                         'provide', 'energy' ) and
           provides_the_function( B.e3.intended_function,
                         'support', 'passenger' ) and
           provides_the_function( B.e4.intended_function,
                         'change', 'direction' ) and
           provides_the_function( B.e5.intended_function,
                         'provide', 'support' ) and
           drives( B.e2, B.e1 ) and
           supports( B.e5, B.e1) and
           supports( B.e5, B.e2) and
           supports( B.e5, B.e3) and
           supports( B.e5, B.e4) }.
```

Figure 9. Definition of a company-specific design principle.

3.2.3 Defining Application-Specific Design Entities

Company-specific design entities are defined as specializations of the generic model of a design entity which was presented in Figure 6. In every company there are particular properties of parts that are of general interest. Thus, Figure 10, presents a definition of a company-specific design entity called a `raleigh_entity` which has these properties.

```
domain raleigh_entity
    =::= { E: entity(E) and
              exists( E.width    : real ) and
              exists( E.mass     : real ) and
              exists( E.material : raleigh_material ) and
              E.mass = mass_of( E ) }.

domain raleigh_material
    =::= { cfrp, titanium, aluminium, steel }.

function mass_of( raleigh_entity ) -> real
    =::= { E -> 2:  E.material = cfrp,
           E -> 3:  E.material = titanium,
           E -> 5:  E.material = aluminium,
           E -> 10: E.material = steel }.
```

Figure 10. The implementation of a company-specific design entity called `raleigh_entity`.

In this figure it can be seen that the concept of a `raleigh_entity` is defined to be a specialization of the generic notion of an `entity`. The specialization consists of an additional three fields, representing `width`, `mass` and `material`, with an equational constraint for estimating the `mass`. When a company has defined its own pseudo-generic concept of a design entity, it can define a repertoire of company-specific design entities. These company-specific design entities may be specializations, with further additional fields, of the company's pseudo-generic concept of design entity or they may be merely synonyms of it. A set of primitive evaluation functions can be defined in terms of the properties of design entities.

3.2.4 Defining Application-Specific Context Relationships

If a design principle specifies a context relationship between some of its embodiments, that relationship must, ultimately, be realized by some analogous relationship between the sets of design entity instances which realize the embodiments so that the product is configured in a valid way and can be properly evaluated.

For example, the `bicycle` principle defined in Figure 9 specifies that a `drives` relationship must hold between the first two embodiments introduced by the principle, those which embody the functions `provide energy` and `facilitate movement`.

The `drives` relationship between two embodiments is defined in Figure 11. This definition specifies that the `drives` relationship

```
relation drives( embodiment, embodiment )
    =::= { (E1,E2): drives( { X | exists entity( X ):
                                   derives_from( X, E1 ) },
                             { Y | exists entity( Y ):
                                   derives_from( Y, E2 ) } ) }.

relation drives( set of entity, set of entity )
    =::= { (E1s,E2s): exists E1 in E1s, E2 in E2s:
                            drives( E1, E2 ) }.

relation drives( entity, entity )
    =::= { (P,W): pedal_assembly(P) and wheel_assembly(W) and
                  is_in_the_same_scheme_as( P, W ) and
                  !exists chain(C):
                      is_in_the_same_scheme_as( P, C ) and
                      !exists mechanical_interface(M1):
                          M1.entity1 = P.id and
                          M1.entity2 = C.id and
                          M1.relationship = drives and
                      !exists mechanical_interface(M2):
                          M2.entity1 = W.id and
                          M2.entity2 = C.id and
                          M2.relationship = drives }.
```

Figure 11. The meaning of the `drives` context relation between embodiments.

holds between two embodiments if and only if an analogous relationship, also called `drives`, holds between the two sets of entity instances that derive from these embodiments. The `drives` relationship between sets of derived entity instances is defined in terms of yet another analogous relationship, this time between individual entity instances.

The precise realization of the context relationship specified in a principle depends on which design entities are used to realize the embodiments that must satisfy the context relationship. Suppose that a `pedal_assembly` is the design entity used to `provide energy` and a `wheel_assembly` is the design entity used to `facilitate movement`, we can see in Figure 11 the relationship that would have to be satisfied between these two entity instances in order to properly embody the drives context relation.

A `mechanical_interface` (Figure 12) is simply a specialization of the generic notion of an `interface`. It can be seen to be a specialization of a application-specific notion of interface, called a `raleigh_interface`, which is a specialization of the generic notion of `interface`.

```
domain mechanical_interface
 =::= { S: raleigh_interface(S) and S.type = mechanical and
           exists( S.relationship : mechanical_relationship ) }.

domain mechanical_relationship
 =::= { controls, drives, supports }.

domain raleigh_interface
 =::= { I: interface(I) and
           exists( I.type : raleigh_interface_type ) }.

domain raleigh_interface_type
    =::= { spatial, mechanical }.
```

Figure 12. Modeling company-specific interfaces.

4 CASE-STUDY

In this section an example of how product structures can be synthesized as schemes will be presented. The example illustrates a very simplified instance of product structuring for configuration. However, the approach presented in this paper has been validated in real-world design domains such as mechatronics, optical systems and electronics design.

In Figure 13 an illustration of the various means contained in an example design knowledge-base is presented. This knowledge-base comprises one design principle, called *bicycle*, and a number of design entities, such as a *wheel assembly* and a *saddle.* The set of behaviors for each means in the knowledge-base are presented under the icon representing the means.

In Figure 14 an instance of the design principle *bicycle*, called *bicycle 1*, has been used to embody the function *provide transport.* This

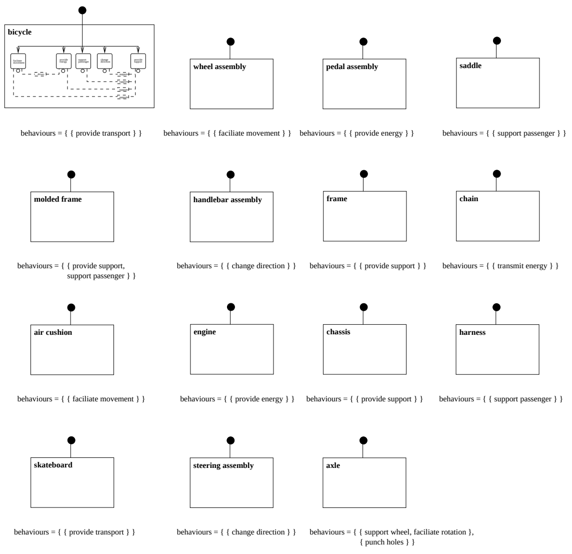

bicycle
faciliate movement
provide energy
support passenger
change direction
provide support
drives
supports
supports
supports
supports
behaviours = { { provide transport } }
wheel assembly
behaviours = { { faciliate movement } }
pedal assembly
behaviours = { { provide energy } }
saddle
behaviours = { { support passenger } }
molded frame
behaviours = { { provide support, support passenger } }
handlebar assembly
behaviours = { { change direction } }
frame
behaviours = { { provide support } }
chain
behaviours = { { transmit energy } }
air cushion
behaviours = { { faciliate movement } }
engine
behaviours = { { provide energy } }
chassis
behaviours = { { provide support } }
harness
behaviours = { { support passenger } }
skateboard
behaviours = { { provide transport } }
steering assembly
behaviours = { { change direction } }
axle
behaviours = { { support wheel, faciliate rotation }, { punch holes } }

Figure 13. The means contained in an example design knowledge-base and their possible functionalities.

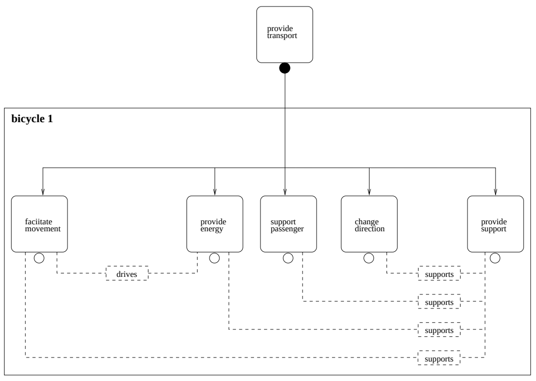


Figure 14. Using a design principle to embody the functional requirement.

design principle introduces the need for five more functions to be embodied. The designer must now select means for embodying each of these functions. The presence of these additional embodiments is due to the constraint-based description of the bicycle design principle.

In Figure 15 the designer selects the *wheel assembly* design entity to embody the function *facilitate movement.* This introduces an instance of this means, called *wheel assembly 1*, into the scheme. As the designer introduces design entities into the scheme the context relations that exist between the function embodiments must be considered. However, since there is only one design entity in the scheme presented in Figure 15 no context relations are considered at this point in the scheme's development.

In Figure 16 the designer has chosen to embody the function *provide energy* with the *pedal assembly* design entity. This introduces an instance of this means, called *pedal assembly 1*, into the scheme. Since the *drives* context relation must exist between the embodiments of the functions *facilitate movement* and *provide energy*, this caused, in addition to the existence of the design entities *wheel assembly 1* and *pedal assembly 1*, the introduction of an instance of the *chain* design entity, called *chain 1*. Both of these interfaces are used, along with *chain 1*, to embody the *drives* relation that should exist between *wheel assembly 1* and *pedal assembly 1*.

Figure 17 shows the state of the scheme after the designer has cho-

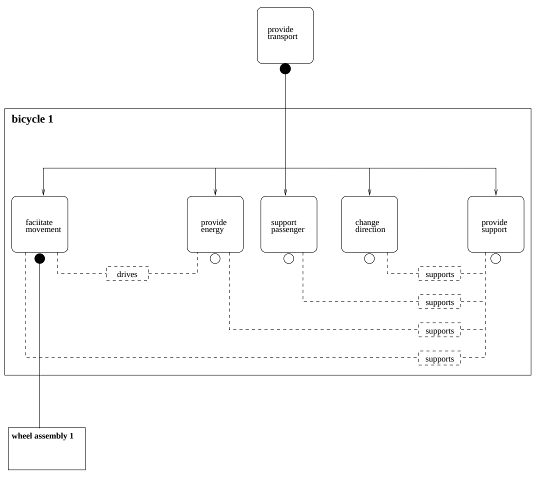


Figure 15. Using a design entity to embody a function in the scheme.

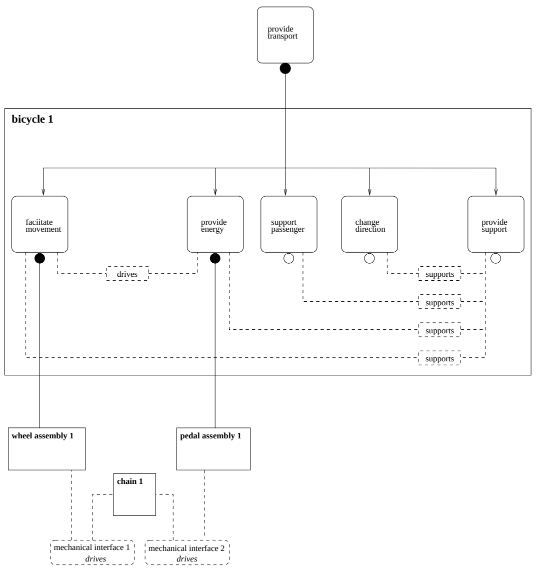


Figure 16. The effect of a context relation on the configuration of design entities.

sen to embody the function *support passenger* with the design entity *saddle*, the function *change direction* with the design entity *handlebar assembly* and the function *provide support* with the design entity *frame*. Due to the *bicycle* design principle, a context relation called *supports* must exist between the embodiment of the function *provide support* and the embodiments of each of the functions *facilitate movement*, *provide energy*, *support passenger* and *change direction*. Each of these context relations is embodied by a *mechanical interface* that defines a *supports* relationship. The details of these mechanical interfaces that define a *supports* relationship will be specified in detail during configuration. Since all the functions have been embodied in the scheme presented in Figure 17, it can be regarded as a product structure which can be be used as the basis for supporting interactive configuration through which a human user will detail parts for the design entities and the interfaces between them. In making these decisions the designer must ensure that the various constraints that are imposed on her due to the design specification or the design

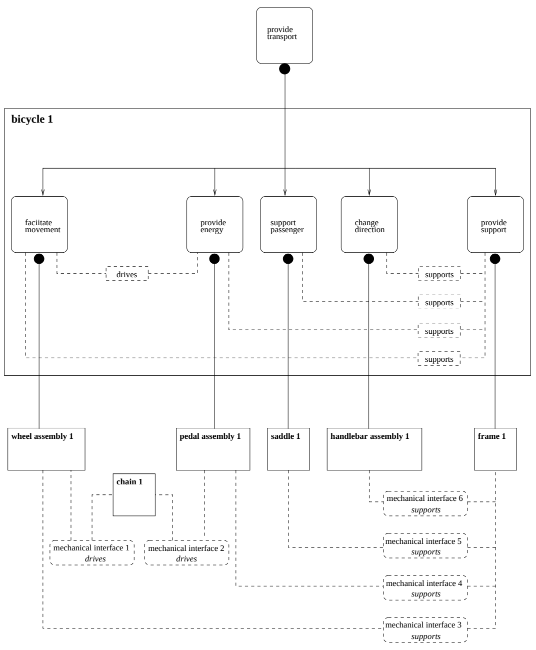


Figure 17. An example scheme configuration.

knowledge-base must be satisfied.

5 CONCLUSION

This paper presents a constraint-based approach to supporting the synthesis of alternative product structures for configuration. The approach is based upon an expressive and general technique for modeling: the design knowledge which a designer can exploit during a design project; the life-cycle environment which the final product faces; the design specification which defines the set of requirements that the product must satisfy; and the alternatives structures that are developed by the designer.

REFERENCES

[1] Mogens Myrup Andreasen. The Theory of Domains. In *Proceedings of Workshop on Understanding Function and Function-to-Form Evolution*, Cambridge University, 1992.

[2] James Bowen and Dennis Bahler. Frames, quantification, perspectives and negotiation in constraint networks in life-cycle engineering. *Artificial Intelligence in Engineering*, 7:199–226, 1992.

[3] Amaresh Chakrabarti and Lucienne Blessing. Guest editorial: Representing functionality in design. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):251–253, 1996.

[4] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumpter. Consistency-based diagnosis of configuration knowledge-bases. In *Proceedings of the 14h European Conference on Artificial Intelligence (ECAI'2000)*, pages 146–150, 2000.

[5] Eugene C. Freuder, Chavalit Likitvivatanavong, and Richard J. Wallace. A case study in explanation and implication. In *In CP2000 Workshop on Analysis and Visualization of Constraint Programs and Solvers*, 2000.

[6] Eugene C. Freuder and Barry O'Sullivan. Generating tradeoffs for interative constraint-based configuration. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming – CP-2001*, November 2001.

[7] Barry O'Sullivan. *Constraint-Aided Conceptual Design*. PhD thesis, Department of Computer Science, University College Cork, Ireland, July 1999. (Also published by Professional Engineering Publishers, Engineering Research Series, 2001).

A Student Advisory System: a configuration problem for constraint programming [1]

Kevin McDonald and **Patrick Prosser** [2]

Abstract. Today, many universities are opting for modular degree programmes. Such modular courses provide greater flexibility for students, allowing them to learn about subjects previously inaccessible to them. Such a system is naturally complex. Modules may feature pre and co-requisites and may run over differently periods of times (and have different credit values). General university requirements will need to be met by students to continue their studies.

Students themselves may further complex the process by explicitly wanting to take or avoid modules. They may require a general overview to see what options are available to them, such as the different routes to a particular degree. The University of Glasgow currently has no automated process to help with this. To find answers students may need to visit several different faculties to investigate possible module selections. Students may have to spend large periods of time with their adviser manually working through the university course catalogue finding what modules are and are not available to them. This paper describes our initial efforts in applying constraint programming to this problem.

1 Introduction

Students design their own degrees. A university degree (for example Computer Science) is typically composed of a set of modules, each corresponding to a specific subject, such as databases, algorithms and data structures, communications, etc. Each of these modules is worth a certain amount of credits. To progress from one year to the next a student must accumulate a minimum amount of credits. In a year of study there will be a set of modules. Typically, a subset of these will be core modules that must be taken. Additional modules must then be selected to achieve the minimum number of credits. This selection may then influence subsequent modules that can, and cannot, be taken in later years. For example, a student cannot take a third year course on algorithms if she has not taken a second year course on discrete mathematics i.e. discrete mathematics is a pre-requisite. Similarly, certain modules must be taken together i.e. they are co-requisites. Added to this, there is a limit to the number of modules a student can take in any year, and sometimes in any term.

Students are then faced with a daunting task. They may decide that they want a certain degree, say Software Engineering, yet there are certain modules that they do and do not want to take, and some terms that they want to minimise the number of modules that they take (maybe so that they can complete a year's project). What modules should they take, and what are the consequences? Much time can be spent by advisors of study assisting students in deciding what to study, and explaining why certain modules cannot be taken i.e. the advisors help the students design their course. Although all this information is available in the university handbook, it may require an expert to interpret it.

What we have attempted here, is to demonstrate that the task of advising a course of study is essentially a problem of design, and that a constraint based model is most appropriate. We demonstrate this by presenting the 4th year curriculum from our department, an encoding of this in the Choco constraint programming toolkit, and sample queries that a student may ask of such a model.

2 A Fourth Year Problem

The 4th year of study is split across two semesters. All students must do an individual final year project (proj) and complete the module on professional issues (pi). Students have then to take 8 other modules, selected from the following options:

1. Formal Methods (fm) semester 1
2. Information Retrieval (ir) semester 1
3. Security & Cryptography (sc) semester 1
4. Advanced Communications (ac) semester 2
5. Artificial Intelligence (ai) semester 2
6. Algorithmics (al) semester 2
7. Computer Architecture (ca) semester 1
8. Databases & Information Systems (dbis) semester 1
9. Design & Evaluation of Multimedia Systems (dems) semester 2
10. Issues in Collaborative & Distributed Systems (hci) semester 1
11. Modelling Reactive Systems (mrs) semester 2
12. Neural Computing (nc) semester 2
13. Network Communications Technology (nct) semester 1
14. Requirements Engineering & Re-engineering (rer) semester 2
15. Safety Critical Systems (scs) semester 1
16. Synthetic Graphics (sg) semester 2

In the first semester a student must take 4 or 5 modules. The semester 1 module Network Communications Technology (nct) is a pre-requisite for the Advanced Communications (ac) module in semester 2.

We represent the above curriculum as a constraint satisfaction problem [4] using the Choco toolkit [1]. This is shown in Figure 1. The Choco function, level4(), delivers an object representing a constraint satisfaction problem composed of integer variables and constraints. Each module is a 0/1 variable, with a value of 1 if taken, 0 otherwise. The two compulsory modules, Professional Issues (pi) and Individual Project (proj), are represented for completeness so that a student is aware that they must be taken (i.e. assigned a value of 1). The constraint on line B represents the pre-requisite: Network

[1] This work was supported by EPSRC research grant GR/M90641.

[2] Computing Science, Glasgow University,17 Lilybank Gardens, Glasgow G12 8RZ, Scotland, email: pat@dcs.gla.ac.uk

```
[level4() : Problem
 -> let pb   := makeProblem("Level 4",20),
        proj := makeIntVar(pb,"proj",0,1),
        pi   := makeIntVar(pb,"pi",0,1),
        fm   := makeIntVar(pb,"fm",0,1),
        dbis := makeIntVar(pb,"dbis",0,1),
        hci  := makeIntVar(pb,"hci",0,1),
        scs  := makeIntVar(pb,"scs",0,1),
        ca   := makeIntVar(pb,"ca",0,1),
        ir   := makeIntVar(pb,"ir",0,1),
        nct  := makeIntVar(pb,"nct",0,1),
        sc   := makeIntVar(pb,"sc",0,1),
        al   := makeIntVar(pb,"al",0,1),
        rer  := makeIntVar(pb,"rer",0,1),
        mrs  := makeIntVar(pb,"mrs",0,1),
        ai   := makeIntVar(pb,"ai",0,1),
        nc   := makeIntVar(pb,"nc",0,1),
        sg   := makeIntVar(pb,"sg",0,1),
        dems := makeIntVar(pb,"dems",0,1),
        ac   := makeIntVar(pb,"ac",0,1),
        must := list(proj,pi),
        sum1 := makeIntVar(pb,"sum1",list(4,5)),
        sum2 := makeIntVar(pb,"sum2",list(3,4)),
        sem1 := list(fm,dbis,hci,scs,ca,ir,nct,sc),
        sem2 := list(al,rer,mrs,ai,nc,sg,dems,ac)
    in (post(pb, sumVars(must) == 2),           // A
        post(pb, implies((ac == 1),(nct == 1))), // B
        post(pb,sumVars(sem1) == sum1),         // C
        post(pb,sumVars(sem2) == sum2),         // D
        post(pb,sum1 + sum2 == 8),              // E
        pb)]
```

Figure 1. level4(), a constraint programming encoding of the curriculum design problem for 4^{th} year Computer Science at Glasgow University

Communications Technology (nct) is a pre-requisite for Advanced Communications (ac). The constraint on line C guarantees that either 4 or 5 modules are taken in the first semester, and the constraint on line D guarantees that either 3 or 4 modules are taken from the second semester. The final constraint E ensures that 8 modules are taken in total. Some of these constraints might initially appear superfluous, but as we will soon see, they are there to allow more interesting queries by the user.

3 Making Choices: an example

The above problem is not *solved* in the conventional sense; instead a user interact with it. The interaction involves enforcing a decision and then seeing the consequences of this, asking for an explanation as to why certain choices are forced upon the user, or why certain choices are not available. We now present a typical sequence of decisions and queries.

Assume we have created the problem (i.e. p:Problem := level4()), and that we have (male) student X. X wants to get started on his project and reckons that he might do well to lighten his load in the first semester. In Choco, we create a new world (i.e. world+()), set sum1 to 4 (i.e. setVal(sum1,4)), and propagate this through the problem (i.e. propagate(p)). This will set sum2 to 4, forcing the student to take 4 modules in the second semester. Note that in creating a new world, we can manually retract the most recent decision by backtracking via the function call world-(), and in the extreme we can return to our initial problem state via the call world=(0).

Student X does not want to take the first semester module Network Communication Technology (nct). Again, we create a new world, now moving to world 2. We set variable nct to 0 (i.e. setVal(nct,0)) and propagate this decision. Since Network Communication Technology is a pre-requisite for Advanced Communication (ac) the variable ac is set to 0 via propagation. Consequently, student X now has a reduced set of options in the second semester.

The above steps can be performed quite easily within the Choco interpreter with just a handful of functions, for selecting and setting variables. That is, as a proof of concept we need not develop a user interface, but merely interact via the interpreter. This might not be unreasonable considering that this model would only be used by 4th year Computer Scientists. However, if and when we move to a more complex environment with more naive users, we expect that a good user interface will be essential.

We could have encoded the above problem differently. In particular, we could have had a constraint stating that 4 or 5 modules must be taken in the first semester i.e. sumVars(sem1) == 4 OR sumVars(sem1) == 5, and similarly that 3 or 4 modules must be taken in the second semester i.e. sumVars(sem2) == 3 OR sumVars(sem2) == 4. However, this would not have allowed student X to make the decision that he will take 4 modules in semester 1. By doing away with the variable sum1, we no longer allow the student to make the strategic decision to spread his study load evenly over the two semesters. Clearly, our choice of model influences the kinds of decisions that a user can make.

4 Giving Explanations

In [2] Junker presents a simple and elegant method for delivering explanations for conflicts. Assume we have a sequence of decisions $S = d_1, d_2, ..., d_n$, where d_n is our last decision before we detect a conflict. Therefore we know that d_n must be one of the culprits. But what other decisions might be involved? Junker proposes that we retract all our decisions and then enforce d_n. If this results in a contradiction we have an explanation, i.e. d_n on its own. If this is not the case we then attempt to make the sequence of decisions $S \backslash d_n$, up to conflict. Assume that on making decisions $d_1 ... d_i$ we again have a conflict. We can then be sure that decisions d_n and d_i together are a subset of the culprit decisions. We then repeat this process, making decisions d_n and d_i, and then the sequence of decisions $S \backslash \{d_n, d_i\}$, again up to conflict, always adding the last decision that fails to the set of culprits. This set of culprits is then a sound and minimal explanation[3].

Junker's technique can be easily extended to cover the situation where we want to determine why propagation sets a specific variable to a specific value i.e. why student X must take a given module or cannot take a given module. We modify the above procedure such that rather than stopping when a contradiction is detected, we stop when a specified variable is set to a specified value. In fact, we can generalise even further, producing an explanation for the removal of a value, a set of values, the setting of a variable, etc.

We use this procedure to deliver explanations. We record all decisions made by the user in a history list H. When a user asks for an explanation, we return to our initial problem via the Choco function call world=(0). This returns us to the first world, where no decisions have been made. We then use the above procedure on the list H building up the list of culprits.

5 Future Work and Conclusion

Within this department, there have been an number of failed attempts at producing a system that can be used to advise students on a course

[3] However, there may be many other explanations.

of study. These failed systems tend to be dominated by attempts to capture the student handbook in a data base and allow access to it via a user interface. Essentially, such systems fail because they do not capture the dynamic effects of decision making. In this project[4] our goal has been to produce a convincing demonstration of constraint programming as a solution to this problem. Our goal was not to produce a fully fledged system, but rather to produce a proof of concept. We believe that we have done that.

Clearly, the above test case (4th year Computer Science) is very small. Attempting to extend this to cover a faculty, let alone an entire university, is a huge task (for example, see van der Linden's PhD thesis [5]). However, we should expect that some day it will become a necessity, especially so as universities become more involved in distance learning. One current example is lifelong learning in the Open University. It is our intention to tackle a larger problem and develop a user interface. We should then be in a position to field the system and evaluate it. In addition, we might also consider higer levels of consistency. At present, only arc-consistency is established when making a decision. It might be interesting to investigate problems where higher levels of consistency are required, i.e. problems where illegal decisions are less obvious.

So far, we have not needed to use techniques from dynamic constraint satisfaction [3]. However, as the model becomes larger and richer we expect this will be a necessity. We are pleasantly surprised at the simplicity and effectiveness of Junker's explanation technique; this fits well with the above problem.

ACKNOWLEDGEMENTS

We would like to thank the Computing Science department of Glasgow University for supporting this final year student project, and in particular Alison Mitchell for patiently explaining the problem to us. We would also like to thank the OCRE project team for their help with Choco.

REFERENCES

[1] CHOCO. http://www.choco-constraints.net/ home of the choco constraint programming system.

[2] Ulrich Junker, 'Quickxplain: Conflict detection for arbitrary constraint propagation algorithms', in *IJCA'01 workshop on Modelling and Solving Problems with Constraints*, pp. 81–88, (2001).

[3] S. Mittal and B. Falkenhainer, 'Dynamic constraint satisfaction problems', in *Proceedings of AAAI-90*, pp. 25–32, (1990).

[4] Edward Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[5] Janet van der Linden. An approach to dealing with non-standard constraint satisfaction problems. PhD Thesis, Oxford Brookes, 2000.

[4] ... a final year project carried out by the first author and supervised by the second.

Problem Decomposition in Configuration

Diego Magro, Pietro Torasso and Luca Anselma [1]

Abstract.

In the present work the issue of decomposing a configuration problem is approached in a framework where the domain knowledge is represented in a structured way by using a KL-One like language and *whole-part relations* play a major role in defining the structure of the configurable objects. The representation formalism provides also a constraint language for expressing complex relations among components and subcomponents.

The paper presents a notion of *boundness* among constraints which assures that two components not involved in a same set of *bound* constraints can be independently configured. The computation of *boundness* among constraints is the basis for partitioning constraints associated with each component to be configured. Such a partitioning induces a decomposition of the configuration problem into independent subproblems.

Both a *recursive* and a *non recursive* decomposition strategies are presented and the savings in computational time and reduction in search space are shown in the domain of PC configuration.

1 Introduction

In recent years configuration has attracted a significant amount of attention not only from the application point of view but also from the methodological one [11]. In particular, logical approaches such as [12, 4] and approaches based on CSP have emerged [8, 2, 10, 13]. In CSP approaches, configuration can exploit powerful constraint problem solvers for solving complex problems. From the other hand, logical approaches make use of a more explicit and structured representation of the entities to be configured (e.g. [7]). Logical approaches seem to offer significant benefits when interaction with the user (e.g. [6]) and explainability of the result (or failure) are major requirements.

Configuration, as many other tasks, can be computationally expensive; therefore, the idea of problem decomposition looks attractive since, from the early days of AI, problem decomposition has emerged as one of the most powerful mechanisms for controlling complexity. Ideally, the solution of a complex problem should be easily assembled by combining the solutions of the subproblems the initial problem has been decomposed into. Unfortunately, in many cases it is not obvious at all how to decompose the problem into a set of non-interacting subproblems.

In the present work the issue of decomposing a configuration problem is approached in a framework where knowledge about the entities is represented in a structured way by using a KL-One like language augmented with a constraint language for expressing complex inter-role relations (see section 2 for a summary of the representation language). Partonomic relations provide the basic knowledge for decomposing the configuration problem. In fact, two subparts involved into two partonomic relations can not be independently configured if there is at least a constraint that links them together. For this reason we have introduced a notion of boundness among constraints (section 3) which assures that two components not involved in a same set of *bound* constraints can be independently configured.

Section 4 provides a high-level description of the configuration algorithm and of the decomposition strategy, while an example of an application of the algorithm is shown in section 5. Section 6 reports a preliminary experiment showing encouraging results as concerns the computational effort. A discussion is reported in section 7.

2 The Conceptual Language

In the present paper the language $\mathcal{FPC}$ [5] is adopted to build the conceptual model of the configuration domains. In $\mathcal{FPC}$ (Frames, Parts and Constraints), there is a basic distinction between *atomic* and *complex* components. *Atomic components* are described by means of set of features characterizing the component itself, while *complex components* can be viewed as structured entities whose characterization is given in term of subparts which can be complex components in their turn or atomic ones. $\mathcal{FPC}$ offers the possibility of organizing classes of (both atomic and complex) components in *taxonomies* as well as the facility of building *partonomies* that (recursively) express the *whole-part relations* between each complex component and any one of its subcomponents. Configuration of an atomic component involves the selection of appropriate values for each feature characterizing the component, while the configuration of a complex component involves a proper assembly of its subparts by using both atomic and complex components. The configuration process has to take into account the *constraints* restricting the set of valid combinations of components and subcomponents. These constraints can be either specific to the modeled domain or derived from the user's requirements.

Frames and Parts. Each *frame* represents a *class* of components (either *atomic* or *complex*) and it has a set of *member slots* associated with it. Each slot represents a *property* of the components belonging to the class and it can be of type either *partonomic* or (alternatively) *descriptive*. Any slot p of a class C is described via a value restriction D(that can be another class or a set of values of a predefined kind) and a number restriction (i.e. an integer interval $[m,n]$ with $m \leq n$), as usual in the KL-One like representation formalisms. A slot p of a class C with value restriction D and number restriction $[m,n]$ captures the fact that the property p for any component of type C is expressed by a (multi)set of values of type D whose cardinality belongs to the interval $[m,n]$.

Partonomic slots are used for capturing the *whole-part relation* between a complex component and a part of its. In $\mathcal{FPC}$ this relation

[1] Dipartimento di Informatica, Università di Torino. Corso Svizzera 185; 10149 Torino; Italy.
e-mail: {magro,torasso}@di.unito.it

is assumed to be asymmetric and transitive.Formally, any partonomic slot p of a class C is interpreted as a relation from C to its value restriction D such that $(\forall c \in C)(m \leq |p(c)| \leq n))$, being $[m,n]$ its number restriction; the meaning is straightforward: any complex component of type C has from a minimum of m up to a maximum of n parts of type D via a whole-part relation named p.

In the following we restrict our attention to partonomic slots since they represent the basic knowledge for problem decomposition.

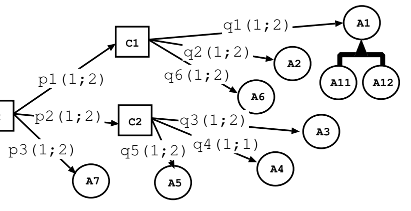
Figure 1 contains a toy conceptual model that we use here as an example. Each rectangle represents a class of complex components, each oval represents a class of atomic components and any thin solid arrow corresponds to a partonomic slot. In the figure, it is stated, for instance, that C is a class of complex components and the partonomic slot $p1$ specifies that each instance of C has to contain one or two (complex) components of type $C1$; while the partonomic slot $p3$ states that any instance of C has to contain one or two (atomic) components of type $A7$.

In any conceptual model, a *slot chain* $\gamma = \langle p_1, \ldots, p_n \rangle$, starting in a class C and ending in a class D is interpreted as the relation composition $p_n \circ p_{n-1} \circ \ldots \circ p_1$ from C to D. The chain represents the subcomponents of a complex component $c \in C$ via the *whole-part* relations named $p_1, \ldots, p_n$. In figure 1, for example, $\langle p1, q1 \rangle$ denotes the subcomponents (of type $A1$) of each instance of C through the partonomic slots $p1$ and $q1$. Similarly, a set of slot chains $R = \{\gamma_1, \ldots, \gamma_m\}$ (where each γ_i starts in C and ends in D_i) is interpreted as the relation union $\bigcup_{i=1}^{m} \gamma_i$ from C to $\bigcup_{i=1}^{m} D_i$.

Besides the partonomies, also the taxonomies are useful in the conceptual models. In figure 1 the subclass links are represented by thick solid arrows. In that toy domain we assume that each class of atomic components Ai is partitioned into two subclasses $Ai1$ and $Ai2$. Only the partitioning of $A1$ into $A11$ and $A12$ is reported in figure.

Constraints. A set (possibly empty) of *constraints* is associated to each class of complex components. These constraints allow one to express those restrictions on the components and the subcomponents of the complex objects that can't be expressed by using only the frame portion of $\mathcal{FPC}$, in particular the inter-slot constraints that cannot be modeled via the number restrictions or the value restrictions.

Each constraint cc associated to C 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 complex component $c \in C$, if c satisfies α then it must satisfy β. It should be clear that if $\alpha = true$, then, for each $c \in C$, β must



CONSTRAINTS

```
Associated with C:
[co1]({<p1,q1>})(1;1) ==> ({<p2,q5>})(1;1)
[co2]({<p1,q2>})(1;1) ==> ({<p2,q3>})(1;1)
[co3]({<p2,q3>})(1;1) ==> ({<p2,q4>})(2;2)
[co4]true ==> ({<p1,q1>})(in A11 (1;4))
[co5]({<p3>})(1;1) ==> ({<p3>})(in A71)
Associated with C1:
[co6]({<q1>})(1;1) ==> ({<q6>})(in A61)
```

Figure 1. A toy conceptual model

always hold, while if $\beta = false$, then, for each $c \in C$, α can never hold.

In the following we present only a simplified version of some predicates available in $\mathcal{FPC}$. For a more complete description of them see [5].

Let $R = \{\gamma_1, \ldots, \gamma_m\}$, where each $\gamma_i = \langle p_{i_1}, \ldots p_{i_n} \rangle$ is a slot chain starting in a class of C complex components. For any $c \in C$, $R(c)$ denotes the values of the relation R computed for c.

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) $(R)(inI)$. $c \in C$ satisfies the predicate *iff* $R(c) \subseteq I$, where I is a union of classes in the conceptual model.

3) $(R)(inI(h;k))$. $c \in C$ satisfies the predicate *iff* $h \leq |R(c) \cap I| \leq k$, where h, k are non negative integers with $h \leq k$ and I is a union of classes in the conceptual model.

For example, the constraint $co5$ states that if only one component playing the partonomic role $p3$ is present in a configuration of an object of type C, then this component must be of type $A71$. The constraint $co4$ states that there must be from 1 up to 4 subcomponents of type $A11$ in each instance of C (through the partonomic slots $p1$ and $q1$).

3 Bound and Unbound Constraints

Given this framework, configuring a complex object of type C means to completely determine an instance c of C in which all the partonomic slots of C are instantiated and each direct component of c is completely configured too. c has to respect both the conceptual model (number and value restrictions imposed by the taxonomy and the partonomy as well as the constraints associated with the classes of components involved in c) and the user's requirements.

Configuring a complex component by taking into consideration only its taxo-partonomical description would be a straightforward activity. In fact, for any well formed model expressed in $\mathcal{FPC}$ in which no constraints are associated with any class, a configuration respecting that model would always exist. A simple algorithm could find it without any search and by simply starting from the class of the target object, considering each slot p of that class and, for it, choosing its cardinality, i.e. choosing the number of components playing the partonomic role p to introduce into the configuration, and the type for each such component. This process must be recursively repeated for each complex component introduced in the configuration, until all the atomic ones are reached. In this process the algorithm needs only to respect the number and the value restrictions of the slots, since the choices made for one partonomic slot do not interfere with the choices for another partonomic slot.

Unfortunately, this is not realistic. The conceptual model usually contains complex constraints that link together different slots. In this more realistic situation a solution can not be found generally without any search and by making only a set of local choices.

Moreover, the requirements usually imposed by the user to the target artifact further restrict the set of legal configurations. This means that the search for a configuration is not guaranteed to be fruitful any more. In fact, even assuming the consistency of the conceptual model, the user's requirements could be inconsistent w.r.t. it and in such a case no configuration following the model and satisfying the requirements exists.

Therefore, in general, the task of solving a configuration problem can be rather expensive from a computational point of view. As we have mentioned above, in $\mathcal{FPC}$ framework this is mainly due to the constraints (both those that are part of the conceptual model and those imposed as user's requirements) that link together different components and subcomponents. In these situations a choice made for a component during the configuration process might restrict the choices actually available for another one, possibly preventing the latter to be fully configured. In such cases the configurator needs to revise some decision that it previously took and to explore a different path in the search space. Usually, in real cases the search space is rather huge and many paths in it do not lead to any solution.

However, in many cases it does not happen that every constraint interacts with each other and the capability of recognizing the sets of (potentially) interacting constraints can constitute the basis for decomposing a problem into independent subproblems.

In principle, once a problem has been decomposed into a set of independent subproblems, these could be solved concurrently and with a certain degree of parallelism, potentially reducing the overall computational time. However, also a sequential configuration process can take advantage of the decomposition. In fact, at least for large configuration problems, solving a set of smaller subproblems is expected to be easier than solving the original one. Moreover, if two subproblems are recognized to be independent, the configurator is aware that no choice made during the configuration process of the first one needs to be revised if it enters a failure path while solving the second one.

To be effective, the task of recognizing the decomposability of a problem (and of actually decomposing it) should not take too much time w.r.t. the time requested by the whole resolution process.

In our approach, the partonomic knowledge can be straightforwardly used in recognizing the interaction among constraints (with an acceptable precision) and in defining a way of decomposing a configuration problem into independent subproblems.

With this aim, we introduce the *bound* relation among constraints, which is based on the *exclusiveness assumption on parts*, stating that, in any configuration, a component can not be a direct part of two different (complex) components, neither a direct part of a same component through two different whole-part relations.

Intuitively, two constraints are *bound* iff the choices made during the configuration process in order to satisfy one of them *can* interact with those actually available for the satisfaction of the second one. If c is a complex component in a configuration, the **bound relation** $\mathcal{B}_c$ is defined in the set $CONSTRS(c)$ of the constraints that c must satisfy, as follows: let $u, v, w \in CONSTRS(c)$. **If** u and v contain both a *same* partonomic slot p of $class(c)$ **then** $u\mathcal{B}_c v$ (i.e. if u and v refer to a same part of c, they are ***directly*** bound in c); **if** $u\mathcal{B}_c v$ **and** $v\mathcal{B}_c w$ **then** $u\mathcal{B}_c w$ (i.e. u and w are bound by ***transitivity*** in c).

It is easy to see that $\mathcal{B}_c$ is an equivalence relation. If U is an equivalence class in the quotient set $CONSTRS(c)/\mathcal{B}_c$, every constraint in U could interact with any other constraint in the same class during the configuration process of c. While, if $V \in CONSTRS(c)/\mathcal{B}_c$ is different from U, it means that in c the constraints in V do not interact in any way with those in U, since the exclusiveness assumption on parts assures that the components and subcomponents of c involved in the constraints in V are different from those referred to by the constraints in U. This means that the problem of configuring c by taking into consideration $CONSTRS(c)$ can be split into the set of independent subproblems of configuring c by considering the set W_i of constraints, for each $W_i \in CONSTRS(c)/\mathcal{B}_c$.

In the next section, we sketch a configuration algorithm that behaves differently on the basis of the value of the parameter k in the procedure $configure$. If $k = 1$ no decomposition is performed, while if $k = 2$ the algorithm tries to *recursively* decompose the problem of configuring each complex component, starting with the target object (*recursive decomposition*). [2]

4 Problem Decomposition

At any stage the current configuration T is represented as a tree. The root TO represents the target object and each node represents a component (either complex or atomic). Each child of a node represents a direct component of it. Given a node n of T, with $class(n)$ we indicate the most specific class (w.r.t. the taxonomy) to which n belongs. The configuration tree T contains also pieces of information useful for the control strategy. In particular: the current component $c\_comp(T)$ (i.e. the one that is being considered currently); the current queue $c\_queue(T)$ containing the direct complex components of $c\_comp(T)$ that still need to be expanded; the information relevant to the constraints holding for $c\_comp(T)$, that is the classes of constraints $classesConstrs(T)$ that still need to be considered and the class of constraints $c\_classConstrs(T)$ that is being considered currently.

Each call to $configure(k, T, CM)$ corresponds to the request of extending the configuration T by configuring the component $c\_comp(T)$ given the constraints in $c\_classConstrs(T)$. CM is the conceptual model (i.e. the $\mathcal{FPC}$ knowledge base) and k defines the decomposition policy. $configure$ returns the pair $\langle Res, Open\rangle$, where Res can be either the extended configuration tree or the $failure$ message and $Open$ is a stack containing the open choices for the configuration of $c\_comp(T)$. At the beginning, $configure$ is invoked (by a *main*) with T containing only the root TO and with $c\_comp(T) = TO$ ($classesConstrs(T)$ and $c\_classConstrs(T)$ are properly initialized as well).

```
PROCEDURE configure(k,T,CM){
 Open := <>;
 WHILE(TRUE){//WHILE-1
     <Expanded_T, L_Open> :=
           insertDirectComponents(T,CM);
    Open := append(Open,L_Open);
    IF(Expanded_T == failure){//IF-1

     IF(Open == <>){RETURN <failure, <> >;
     }ELSE{T := pop(Open);}

    }ELSE{/*Expanded_T =/= failure*/
      T := Expanded_T;
      c := c_comp(T);

      WHILE(c_queue(T) =/= <>){//WHILE-2
       c_comp(T) := extract(c_queue(T));
       IF(k == 1){
       classesConstrs(T) := NoDecompose(T,CM);
       }ELSE{classesConstrs(T) := decompose(T,CM);}
       WHILE(classesConstrs(T) =/= <>)
                       {//WHILE-3
        c_classConstrs(T) :=
                 extract(classesConstrs(T));
        <Extended_T, L_Open> := configure(k,T,CM);
        IF(Extended_T == failure){//IF-2
         - delete from Open all the elements
           labelled with the indentifier of
           c_comp(T);
         IF(Open == <>){//IF-3
          RETURN <failure, <> >;
         }ELSE{
           - Try to complete the configuration
             by properly revising the past
             choices on the basis of Open;
           - RETURN the result;
```

[2] In the first case a much simpler algorithm could be written. Due to space constraints, we describe the two different approaches within a single algorithm.

```
          }//IF-3
        }ELSE{/*Extended_T =/= failure*/
          IF(L_Open =/= <>){
           - label L_Open with the
             c_comp(T) identifier;
           push(L_Open,Open);}
          T := Extended_T;}//IF-2
       }//WHILE-3
       c_comp(T) := c;
      }//WHILE-2
      RETURN <T, Open>;
     }//IF-1
 }//WHILE-1
}//configure
```

The first step performed by $configure$ is the introduction of the direct components of the current component $c\_comp(T)$. This is done by considering each (not yet considered) slot p of the class $class(c\_comp(T))$ and by choosing both the number of the components playing the partonomic role p to be inserted in the configuration and the type for each of them. [3] Any time a direct *complex* component of $c\_comp(T)$ is introduced, it is inserted in $c\_queue(c\_comp(T))$. All the open choices are stored in $Open$. [4] It can happen that in this phase there is no way to insert the direct components of $c\_comp(T)$ into the configuration without violating any constraint. In this case, if no alternative is available in $Open$ the $failure$ message is returned, otherwise an alternative is considered.

The WHILE-2 loop considers each complex direct component of $c\_comp(T)$ that has not been expanded (i.e. configured) yet. Such a component becomes the new current component in T. The way in which the set $classesConstrs(T)$ is computed for this new current component defines the behaviour of the configuration process w.r.t. the decomposition.

If it is computed by the $NoDecompose(T, CM)$ function (i.e. $k = 1$), all the constraints relevant to the current component are considered in a unique chunk and no decomposition is performed, since $NoDecompose(T, CM) = \{I\_Constrs \cup L\_Constrs\}$; where $I\_Constrs$ is the set containing all the current constraints of the parent component of $c\_comp(T)$ (if any) that mention some slot of $class(c\_comp(T))$, and $L\_Constrs$ is the set of constraints associated with $class(c\_comp(T))$ and those expressing the user's requirements for that component. If $k = 2$ $decompose(T, CM) = (I\_Constrs \cup L\_Constrs)/\mathcal{B}_{c\_comp(T)}$ function is invoked.

By making use of the bound relation $\mathcal{B}_c$ among constraints (see section 3), $decompose$ partitions the constraints into classes of bound constraints and returns these classes (i.e. it returns the quotient set $CONSTRS(c\_comp(T))/\mathcal{B}_{c\_comp(T)}$).

One such computed class of constraints is considered at a time (WHILE-3 loop), i.e. a current class of constraints $c\_classConstrs(T)$ is repeatedly extracted from the set $classesConstrs(T)$ and the $configure$ procedure is invoked to configure the current component $c\_comp(T)$ *w.r.t. the current class of constraints* [5]. This means that the problem of configuring the current component w.r.t. the constraints $CONSTRS(c\_comp(T))$ has been split into a set of subproblems, one for each class of constraints contained in $classesConstrs(T)$. These subproblems have been entailed by the partitioning of the constraints $CONSTRS(c\_comp(T))$ into a set of classes. Since any two constraints of two different classes are unbound, the configuration choices made while taking into consideration one of them are independent from those relevant to the second one. Of course, whenever $classesConstrs(T)$ contains a single class, no decomposition is performed. If the called $configure$ procedure succeeds in configuring the component $c\_comp(T)$ w.r.t. $c\_classConstrs(T)$ (ELSE branch of IF-2), the calling one (whose task is to configure the parent component of $c\_comp(T)$) stores the returned stack L_Open of (local) open choices for $c\_comp(T)$ by pushing it in its $Open$ stack. Thus, the elements stored in $Open$ can either be configuration trees or stacks in their turn. In the first case they (explicitly) represent the open choices relevant to the insertion of the direct components. In the second one, they (implicitly) represent the open choices relevant to the configuration of the complex direct components. Each stack pushed in $Open$ is labeled with the identifier of the complex direct component to which it refers. In this way, if the configuration of any direct component w.r.t. a given class of constraints fails (THEN branch of IF-2), all the open choices relevant to the configuration of the *same* component w.r.t. each previously considered class of constraints can be removed easily. Therefore, some useless backtrackings are avoided.

If a failure occurs while configuring a direct component and $Open$ contains some alternatives (ELSE branch of IF-3), a revision of the configuration takes place. If the top of the stack $Open$ contains a configuration tree, this is analogous to the the case of failure while introducing the direct components. If the top of $Open$ is a stack in its turn, the nesting of stacks is used it order to maintain the correspondence between the recursive calls of $configure$ and the partonomic structure of the configuration. We do not enter into the details of such a revision because of space limits.

Before presenting a simple example, we outline that a third decomposition policy can be defined in which only the constraints relevant to the target object TO are partitioned. This can be done by using the $decompose$ function (in the *main*) before calling $configure$ for the first time and then by calling it with $k = 1$. In this way, only the main problem can be decomposed, but the decomposition does not take place in the components and subcomponents of TO (*non recursive decomposition*).

5 An Example

Figure 2 reports five snapshots of a possible configuration process for configuring an object of type C (fig. 1) [6]. In figure 2, each c_i identifies the i^{th} component of type C. The numbers preceding the identifiers express the order in which the components have been introduced into the configuration. We consider the *recursive decomposition* policy.

At the beginning, only the root c_1 is inserted (by the *main*) into the configuration (as current component). The *main* invokes the $decompose$ that partitions the set of constraints for c_1 into the two different classes of *bound* constraints $V_1 = \{co1, co2, co3, co4\}$ and $V_2 = \{co5\}$. The main problem is thus decomposed into two independent subproblems (one relevant to V_1 and the other relevant to V_2). $configure$ is invoked (with $k = 2$) to solve the first subproblem. The direct components $c1\_1$ and $c2\_1$ are inserted and the open choices (i.e. the alternative cardinality 2 for both the slots $p1$ and $p2$) are saved in $Open$.

[3] When we speak about *choices* made while configuring a component we refer exactly to these two kinds of choices.

[4] For simplicity, we can assume that at this point, each time one or more alternatives are available and before actually modifying the configuration, such alternatives are stored in the current configuration tree which is then pushed into L_Open.

[5] It is worth noting that the procedure $configure$ considers at each time only the slots of $class(c\_comp(T))$ appearing in the constraints $c\_classConstrs(T)$.

[6] For simplicity, we do not distinguish between the constraints present in the conceptual model and those representing the user's requirements.

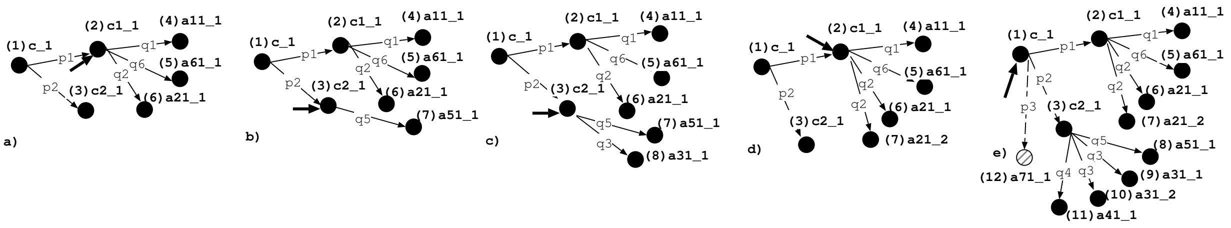


Figure 2. A configuration example

Then, the configuration of the component $c1\_1$ is split into 2 other subproblems, since its constraints (those inherited from c_1 and the local one $co6$) are partitioned into the 2 classes $V_3 = \{co1, co4, co6\}$ and $V_4 = \{co2\}$. It is worth noting that in c_1 the constraint $co2$ belongs to the same class as the constraints $co1$ and $co4$, while in the component $c1\_1$ the constraint $co2$ belongs to a class different from that containing $co1$ and $co4$. In fact, the constraint $co2$ can interact with the constraints $co1$ and $co4$ during the configuration process of c_1; but, $co2$ does not interact with $co1$ or $co4$ while configuring the component $c1\_1$. It is worth noting that at least in the case under examination, the recursive decomposition policy was able to recognize that a set of constraints that are bound in the larger context of the configuration of the target object c_1 can actually be split in the (smaller) context relevant to the configuration of the component $c1\_1$.

The solutions of the two subproblems corresponding to V_3 and V_4 lead to the situation depicted in fig. 2 a (the thick arrow points to the current component: in this case, $c1\_1$). Then (fig. 2 b) the problem of configuring $c2\_1$ is decomposed into 2 subproblems associated with the two classes $V_5 = \{co1\}$ and $V_6 = \{co2, co3\}$. After considering V_5, the problem associated with V_6 is taken into consideration (fig. 2 b). It should be clear that the configuration in fig. 2 c can not be extended in any way satisfying $co3$. Choosing a different type (i.e. $A32$) for the eighth component would not solve the problem. Since this was the only open alternative in the configuration of $c2\_1$ w.r.t. V_6 (in fact, due to $co2$, no alternative cardinality could be chosen for $q3$) the resolution of the subproblem associated with V_6 fails. It is worth noting that the alternatives for the subproblem associated with V_5 are not considered (see the THEN branch of IF-2 in $configure$) and a revision of the configuration of $c1\_1$ takes place (fig. 2 d). In this way, $c2\_1$ can be completely configured and thus the subproblem associated with V_1 is successfully solved. The resolution of the problem associated with V_2 adds the $a71\_1$ component to the configuration (fig. 2 e).

6 Preliminary Results

The domain of PC configuration has been used for testing the impact of the decomposition strategies. We have generated a test set of 153 configuration problems: for each of them we have specified the type of the target object (e.g. a PC for graphical applications) and the requirements that it must satisfy (e.g. it must have a CD writer of a certain kind, it must be *fast enough* and so on).

The goal of the experiment is to compare (w.r.t. the computational effort) among them a configuration strategy without decomposition (i.e. no constraints partitioning in the *main* that calls $configure$ procedure with $k = 1$) (*Strategy* 1), the *non-recursive* strategy that attempts to decompose only the main configuration problem (see section 4) (*Strategy* 2) and the *recursive* one (*Strategy* 3).

The configuration system has been implemented in Java using JDK 1.3 and the experiment has been performed on a Duron 700/Windows 2000 PC with 128 Mbytes RAM. A problem is considered solved iff the configurator provides a solution for it within the time threshold of 360 sec. or if it detects (within the same timeout) that the problem does not admit any solution.

Strategies 1 and 2 both solved the same 150 problems (i.e. their competence was of 98%), while strategy 3 solved all the 153 cases. As regards the computational effort, for the 150 problems solved by all the three strategies, the average computation time was (in msec.) 7735 for strategy 1; 7465 for strategy 2 (i.e. -3.5% w.r.t. strategy 1) and 2257 for strategy 3 (i.e. -69.8% w.r.t. strategy 2); the average number of backtrackings was 770 for strategy 1; 751 for strategy 2 (i.e. -2.5% w.r.t. strategy 1) and 567 for strategy 3 (i.e. -24.5% w.r.t. strategy 2). The benefits of strategy 2 w.r.t. strategy 1 become more evident if we consider the 49 solved problems that were actually decomposable with strategy 2. For these problems, the average CPU time was (in msec.) 3468 for strategy 1 and 2754 (i.e. -20.6%) for strategy 2; the average number of backtrackings was 559 for strategy 1 and 501 for strategy 2 (i.e. -10.4%).

7 Discussion and Future Work

The present paper addresses the problem of partitioning a configuration problem into simpler subproblems by exploiting as much as possible the implicit decomposition provided by the partonomic relations of complex components. The adoption of a structured approach based on a logical formalism plays a major role since the criterion for singling out "unbound" constraints is based on an analysis of the partonomic slots mentioned in the constraints. Moreover, the constraints are associated with each complex component. The configuration algorithm tries to recursively decompose the configuration problem each time it is invoked to configure a component or a subcomponent.

The paper presents a centralized version of the configuration strategy in which the subproblems resulting from a decomposition are sequentially solved. An alternative configuration strategy based on a distributed approach could be conceived: each time a configuration problem is split into a set of independent subproblems, these could be sent to a pool of configurators running in parallel.

Our main motivation for decomposing configuration problems is saving in computational effort. Preliminary results are encouraging in this respect since benefits are present even with a sequential approach to configuration. The preliminary experiments have also shown that the results are influenced by the order in which the partonomic slots are considered. Therefore, it is worth investigating strategies able to suggest convenient order in instantiating partonomic slots. In the following, we will sketch one of such strategies, based on hypergraph cut techniques similar to the one proposed for decomposing the problem of satisfiability testing on large propositional CNF formulas, pre-

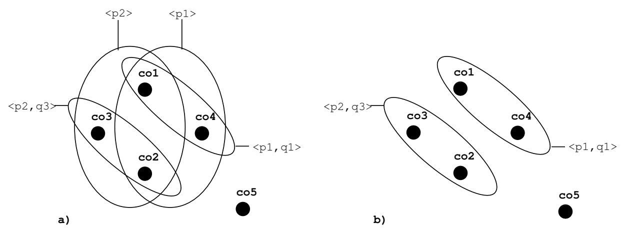


Figure 3. Hypergraphs associated with the constraints for c_1

sented in [9]. [7] In that approach, a hypergraph is associated with a CNF formula F such that each variable of F corresponds to a hyperedge connecting all the clauses in which it occurs. Then, such a hypergraph is used to find an appropriate set V_c of variables allowing two subformulas F_1 and F_2 of F to be individuated such that the problem of testing the satisfiability of F corresponds to the problem of verifying if there is some compatible truth assignment to the variables in V_c that makes F_1 and F_2 satisfiable. For each assignement to the variables in V_c, F_1 and F_2 simplify into two formulas that can be tested independently (see [9] for details).

Following this framework, we can associate a hypergraph to the set $CONSTRS(c)$ that any complex component c in a configuration must satisfy. For example, in figure 3.a the hypergraph associated with the constraints for target object c_1 is depicted. In such a hypergraph, each constraint corresponds to a vertex and two constraints are connected by a same hyperedge iff they are directly bound. In figure 3.a, for instance, the constraints $co1$ and $co2$ belong to a same hyperedge, since they are *directly* bound, while there is no hyperedge containing both $co3$ and $co4$, since they are not *directly* bound. Moreover, two constraints are bound iff they belong to a same connected component in the hypergraph. For example, in figure 3.a, $co3$ and $co4$ are bound (even if not directly), since they belong to a same connected component. Instead, the constraint $co5$ belongs to a connected component different from the one containing all the other constraints, therefore $co5$ is not bound to any other constraint. It should be clear that each connected component of the hypergraph associated with the constraints $CONSTRS(c)$ corresponds to an equivalence class w.r.t. the bound relation $\mathcal{B}_c$. Thus, computing the quotient set $CONSTRS(c)/\mathcal{B}_c$ means computing the set of connected components of the hypergraph associated with $CONSTRS(c)$.

Differently from the hypergraphs introduced in [9], the hyperedges of a hypergraph associated with a set of component constraints in the $\mathcal{FPC}$ framework do not represent simple variables. Instead, each hyperedge represents a structured entity, namely a slot chain providing an explanation of the fact that the constraints belonging to the hyperedge are directly bound. For example, the hyperedge labelled $\langle p2 \rangle$ in the hypergraph of figure 3.a states that the constraints $co1$, $co2$ and $co3$ are directly bound in c_1 by means of the slot chain $\langle p2 \rangle$. Moreover, the hyperedge labelled $\langle p2, q3 \rangle$ states that the constraints $co2$ and $co3$ are directly bound in c_1 by means of the slot $p2$ and that they still remain directly bound (by means of the the slot $q3$) in each direct component of c_1 playing the partonomic role $p2$.

Despite this difference, the decomposition methods presented in [9] suggest the opportunity of investigating a possible improvement of our decomposition technique. Let's consider again the hypergraph in figure 3.a. Removing the hyperedges $\langle p1 \rangle$ and $\langle p2 \rangle$ would increase the number of connected components (figure 3.b), i.e. it would increase the number of equivalence classes in the quotient set $CONSTRS(c\_1)/\mathcal{B}_{c\_1}$. This means that, after having introduced into the configuration the direct components of c_1 through $p1$ and $p2$, the subproblem of configuring the target object c_1 by considering the constraints $V_1 = \{co1, co2, co3, co4\}$ (see section 5) can be further split into two other subproblems corresponding to the two sets of constraints $\{co1, co4\}$ and $\{co2, co3\}$. In fact, these two subproblems can be solved independently, since the only choices that can interact with them are those made for $p1$ and $p2$.

Therefore, as in the case of satisfiability testing of CNF propositional formulas, hypergraph cut techniques can be used to find an appropriate set of hyperedges (i.e. of slot chains) whose deletion from the hypergraph would increase the number of connected components. Such set of slot chains can then be used by the configuration algorithm in order to guide the selection of the slots to be considered at each time and to dynamically split the configuration problems.

Besides the reduction in computational effort, there are other good reasons for investigating decomposition. First of all, in some domains, the configuration problems are distributed by their nature [1]. Moreover, one of the serious problems in configuration is the explanation task [3]. The adoption of a structured approach for modeling the domain knowledge and the introduction of a decomposition module able to specify which constraints involved in the configuration of a given component can interact with each other is a first step for making progresses in the explanation of failure in configuration.

[7] The potential similarity between our approach and the one presented in [9] has been suggested by an anonymous reviewer.

REFERENCES

[1] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, 'Distributed configuring', in *Proc. IJCAI-01 Configuration WS*, pp. 18–24, (2001).

[2] 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).

[3] E. Freuder, C. Likitvivatanavong, and R. Wallace, 'Explanation and implication for configuration problems', in *Proc. IJCAI-01 Configuration WS*, pp. 31–37, (2001).

[4] G. Friedrich and M. Stumptner, 'Consistency-based configuration', in *AAAI-99, Workshop on Configuration*, (1999).

[5] D. Magro and P. Torasso, 'Description and configuration of complex technical products in a virtual store', in *Proc. ECAI 2000 Configuration WS*, pp. 50–55, (2000).

[6] D. Magro and P. Torasso, 'Supporting product configuration in a virtual store', *LNAI*, **2175**, 176–188, (2001).

[7] D. L. McGuinness and J. R. Wright, 'An industrial-strength description logic-based configurator platform', *IEEE Intelligent Systems*, (July/August 1998), 69–77, (1998).

[8] S. Mittal and B. Falkenhainer, 'Dynamic constraint satisfaction problems', in *Proc. of the AAAI 90*, pp. 25–32, (1990).

[9] T. Joon Park and A. Van Gelder, 'Partitioning methods for satisfiability testing on large formulas', *Information and Computation*, (162), 179–184, (2000).

[10] D. Sabin and E.C. Freuder, 'Configuration as composite constraint satisfaction', in *Proc. Artificial Intelligence and Manufacturing. Research Planning Workshop*, pp. 153–161, (1996).

[11] D. Sabin and R. Weigel, 'Product configuration frameworks - a survey', *IEEE Intelligent Systems*, (July/August 1998), 42–49, (1998).

[12] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen, 'Unified configuration knowledge representation using weight constraint rules', in *Proc. ECAI 2000 Configuration WS*, pp. 79–84, (2000).

[13] M. Veron and M. Aldanondo, 'Yet another approach to ccsp for configuration problem', in *Proc. ECAI 2000 Configuration WS*, pp. 59–62, (2000).

A multi-perspective approach for the design of configuration systems

L. Hvam, J. Riis & M. Malis[1]

ABSTRACT
This article presents a procedure for building product models to support the specification processes dealing with sales, design of product variants and production preparation in manufacturing companies. The procedure includes, as the first phase, an analysis and redesign of the business processes that are to be supported by the use of product models. The next phase includes an analysis of the product assortment and the set up of a so-called product master. Finally the product model is designed by using object-oriented modelling.

The procedure has been tested in several companies. This article includes the experiences gained from the most recent project that was carried out at Demex-electric – a Danish company, which manufactures electronic switchboards. The research has been carried out at the Centre for Product Modelling, Department of Manufacturing Engineering at the Technical University of Denmark.

1. INTRODUCTION

This article presents a procedure for building product- and product-related models that support the specification process (figure 1).

The activities in the "specification process" includes an analysis of the customer's needs, design and specification of a product which full-fill the customer's needs and specification of e.g. the products manufacturing, transportation, erection on site and service (specification of the product's life cycle properties). The activities in the specification process are characterised by having a relatively well-defined space of (maybe complex) solutions as a contrast to product development, which is a more creative process.

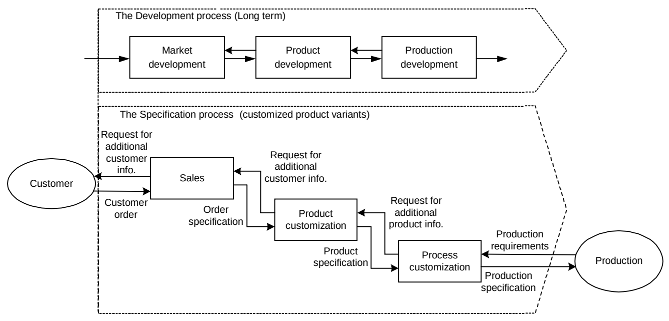


Figure 1. The specification process

Typical goals for the specification process are the ability to find an optimal solution according to the customer's needs, high quality of the specifications, short lead time and a high productivity of the work carried out in the specification process. The typical critical goals for the development process are to derive new original concepts of product designs with improved functionality and life cycle properties, and to reduce time to market for the new product designs. The diversification of tasks and goals in the specification and development processes leads to a separation of the two processes as suggested in figure 1 above.

Today we see numerous examples of projects aiming at implementing product- and product-related models to support the specification processes. Experiences from a considerable number of Danish and international companies show that often these models are constructed without the use of a strict procedure or modelling

[1] Centre for Product Modelling, www.productmodels.org Department of Manufacturing and Management, Technical University of Denmark, Building 423, DK-2800 Lyngby, Denmark.

techniques. As a result of this, the systems are unstructured and undocumented, and therefore difficult or impossible to maintain or develop further.

Thus there is a need for developing a procedure and the associated modelling techniques, which can ensure a proper structure and documentation, so that the systems can be maintained continually and developed further. Experiences also show that the product- and product-related models are not always designed to fit the business processes that they are meant to support. Finally an important precondition for building product models is the fact that products must be designed and structured in a way, which makes it possible to define a general master of the product, from which the customer specific products can be derived.

Consequently, in order to cope with these challenges a procedure for building product models should include: An analysis and redesign of the specification processes in focus, an analysis and eventually redesign/ restructuring of the products to be modelled, and finally, a structured "language" - or modelling technique - which makes it possible to document the product- and product-related models in a structured way.

The procedure suggested here – or part of the procedure - has been tested in several companies, e.g. F.L.Smidth, American Power Conversion (APC), Aalborg industries, NEG-Micon, GEA-Niro and IBM-SMS. The article includes the experiences acquired from the latest project at Demex-electric – a Danish manufacturer of electronic switchboards. The procedure is based on four aspects (figure 2).

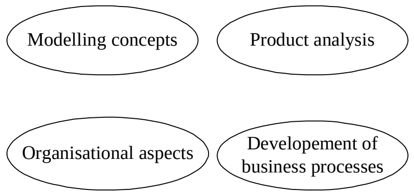


Figure 2. Four aspects incorporated in the procedure.

1. Modelling concepts – based on object-oriented modelling.
2. Product analysis – concerning the transformation of product knowledge into structured and visual information.
3. Organizational aspects – how to organize the development of product- and product-related models and the subsequent use of the models.
4. Development of business processes – how to identify and redesign new business processes supported by product- and product-related models.

2. A PROCEDURE FOR BUILDING PRODUCT MODELS

The procedure contains 7 phases. The starting point of the work is an analysis and redesign of the business processes that will be affected by the product- and product-related models (phase 1). In phase 2 the products are analysed and described in a so-called product master [3]. Phase 3 includes the final design of the product- and product-related models by using the object-oriented modelling techniques. Phases 4 to 7 deals with design, programming, implementation and maintenance of the product models. Phases 3 to 7 are based on the general object-oriented project life cycle.

There may be some overlap and iterations between the individual phases. The procedure is shown in figure 3.

Phase	Description
1	**Process Analysis** Analysis of the existing specification process (AS-IS), statement of the functional requirements to the process. Design of the future specification process (TO BE). Overall definition of the product- and product-related models to support the process. [6],[7]. **Tools:** IDEF0, flow charts, Activity Chain, Model, key numbers, problem matrix, SWOT, list of functional describing characteristics and gap analysis.
2	**Product Analysis** Analysing products and eventually life cycle systems. Redesigning/ restructuring of products. Structuring and formalizing knowledge about the products and related life cycle systems in a product master. **Tools:** List of features and product master [3].
3	**Object-oriented Analysis (OOA)** Creation of object classes and structures. Description of object classes on CRC-cards. Definition of user interface. Other requirements to the IT solution. **Tools:** Use cases, screen layouts, class diagrams and CRC-cards.
4	**Object-oriented Design** Defining and further developing the OOA-model for a specific programming tool.
5	**Programming** Programming the system. Own development or use of standard software.
6	**Implementation** Implementation of the product- and product-related models in the organization. Training users of the system, and further training of the people responsible for maintaining the product- and product-related models.
7	**Maintenance** Maintenance and further development of the product- and product-related models.

Figure 3. A procedure for building product models, [1].

3. CASE STUDY

The procedure has been used for building and implementing a sales configurator in the Danish company Demex-electric. The experiences gained from the process are described in the following sections.

3. 1 CASE COMPANY: DEMEX-ELECTRIC

Demex-electric is a Danish manufacturer of electronic switchboards. It has more than 100 employees and a turnover of approx. 15 million Euro. An example of a switchboard is showed in figure 4.

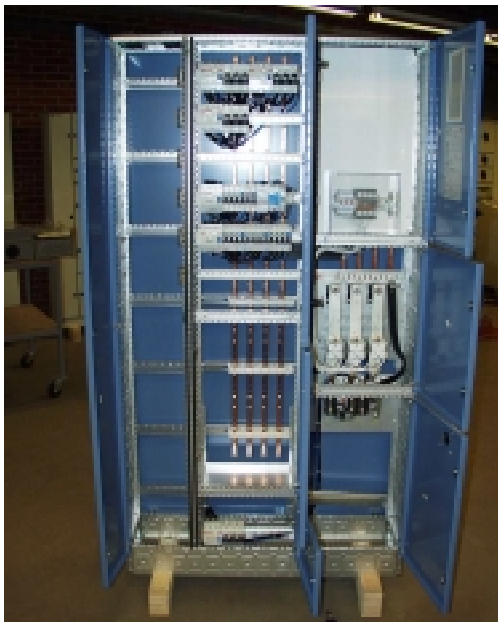


Figure 4. An example of an assembled unit manufactured at Demex.

3. 2 PHASE 1: PROCESS ANALYSIS

As the process is now, the customer indicates, among other things, the power required, the electricity companies that can supply the power, the demands for protection, the switchboard outlets etc. Then Demex starts specifying the switchboard and prepares a list of parts, makes a sketch, calculates the price and writes a quotation letter. When the customer has accepted the offer, the list of parts and the sketch are detailed, and can now form the basis of purchasing and production.

The lead-time for generating quotations is 3 to 5 days. Demex uses 2 to 4 man-hours for each quotation. The process may lead to frequent errors, and often the time necessary for the optimisation of the boards cannot be found. When the future process requirements at Demex were to be set, a SWOT analysis was also made, as shown in figure 5.

As a consequence of this Demex has made an alliance with Solar A/S – a company selling electronic equipment. In Denmark Solar A/S has a turnover of € 250 million, 50 % of the products are sold directly via Solar's homepage.

Opportunities	**Treats**
Sale, quotations and configuration of switchboards on the Internet Improved quality of specifications Reduction of costs of assembly Logistics will be improved Market size DKK 1.5 billion	Many competitors (35) All switchboards are identical seen from a customer point of view Not possible to differentiate Low net profit ratio within this business sector
Strengths	**Weakness**
Knowledge and experience of employees Good image and branding Broad product range	SC-utilisation Size in the market

Figure 5. SWOT-analysis.

In the alliance between Demex-electric and Solar A/S, Solar A/S hosts the system for configuring an electronic switchboard as an integrated part of their homepage for selling electronic equipment. When a customer configures and orders an electronic switchboard, Solar A/S ships the parts needed for building a switchboard to Demex. The switchboards are then assembled and shipped. The future process focuses on ensuring efficient quotations by using the Internet. The lead-time and the consumption of resources, from the generation of quotations until the specifications lie in the production department, are considerably reduced. In connection with the analysis of the existing process at Demex, a number of flow diagrams were made.

With the new product model the company gets a much more structured process flow where the knowledge of Demex regarding construction of switchboards is made available to the customers, and complex calculations can be made very quickly. This is illustrated in figure 6.

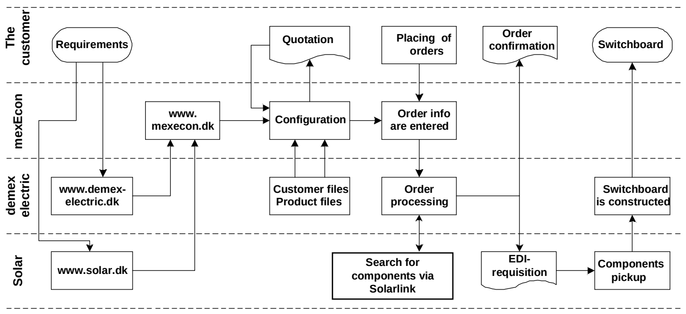


Figure 6. The Future process flow when applying a configurator.

The configurator is named mexEcon. The effects of mexEcon are identified as:

1. Customers can make a quotation in 10 minutes 24 hours a day.
2. Reduction of lead time from 3-4 days to 10 minutes when generating quotes.
3. Possible to optimise a configuration in relation to e.g. materials used and performance of the switchboard.

4. The time used for the specification of switchboards is reduced from 2-4 hours to 10 minutes.

3. 3 PHASE 2: PRODUCT ANALYSIS

In this phase the products to be modeled are analyzed in order to gain an overview of the product families and their structure. The analysis covers the function, structure, properties and variations of the product and the related systems in the product's life cycle. Figure 7 shows a general architecture for describing products including the above-mentioned views.

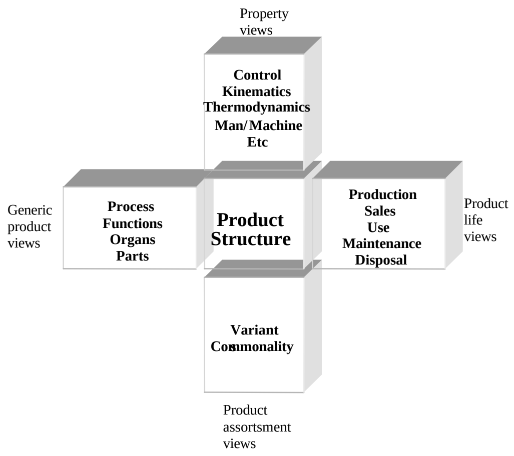


Figure 7. An architecture for describing products [2], [3].

A formal way of describing the product assortment is a so-called product master [3]. A product master consists of two main elements: a generic part-of structure and a generic kind-of structure. Experiences from this case showed that the product master was a very good tool for discussing the product assortment, i.e. where to start modelling in the configuration system, which variants to include, which technologies are stable and which technologies will change etc. The product master is described on a big piece of paper around 3 x 1meters. Once a week (in several months) a meeting was held in order to discuss the products. All the domain experts were present at these meetings. This was mainly to ensure commitment and agreement. A part of the product master is illustrated in figure 8.

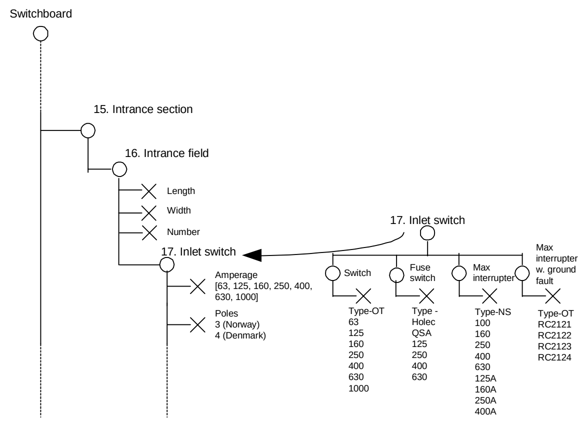

Switchboard
15. Intrance section
16. Intrance field
Length
Width
Number
17. Inlet switch
Amperage [63, 125, 160, 250, 400, 630, 1000]
Poles 3 (Norway) 4 (Denmark)
17. Inlet switch
Switch
Fuse switch
Max interrupter
Max interrupter w. ground fault
Type-OT 63 125 160 250 400 630 1000
Type - Holec QSA 125 250 400 630
Type-NS 100 160 250 400 630 125A 160A 250A 400A
Type-OT RC2121 RC2122 RC2123 RC2124

Figure 8. A part of the product master.

The left side of the product master (the part of structure) includes the modules or parts used in the entire product family. This is due to the aggregation structure in object-oriented modelling. The left side of the product master (kind of structure) includes the parts, which can be exchanged in the product. This is due to the specialisation structure in object-oriented modelling.

3. 4 PHASE 3 & 4: OBJECT ORIENTED ANALYSIS AND DESIGN

OOA is a method used for analysing a given problem domain and the field of application in which the IT system will be used. The purpose of the OOA is to analyse the problem domain and the field of application in such a way that relevant knowledge can be modelled in an IT system. The problem domain is the part of reality outside the system that needs to be administrated, surveyed or controlled. The field of application is the organization (person, department) that is going to use the system to administer, survey or control the problem domain.

The OOA model can be made through the activities described in figure 9 which describes the OOA as consisting of five phases or layers. These layers can be seen as different viewpoints, which together make up the OOA model. Normally the five activities are carried out through a top down approach, but there are no restrictions in that sense. Typically the OOA model will be the result of a number of iterations of the analysis process.

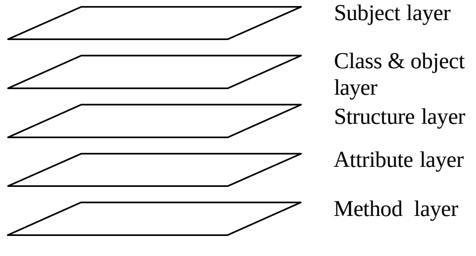


Figure 9. The five layers of OOA modelling [13].

The *subject layer* contains a sub-division of the complete domain, which is to be modelled in different subject areas. In relation to the use of product models, a subject area can for example be a product model or a factory model.

The *class and object layer* contains a list of the classes and objects, which have been identified in the individual subject areas.

The *structure layer* contains the relationships between the objects, i.e. a specification of generalization and aggregation.

The *attribute layer* contains a specification of the information associated with the individual objects, i. e what the objects know about themselves.

The *method layer* contains a description of the individual objects methods (procedures), i. e. what the objects can perform.

The static structure is mirrored in the layers of theme, classes and objects, structure and attributes, while the more dynamic aspects in the model mainly are related to the method layer. The result of the OOA can be illustrated in a class diagram and on CRC-cards (Class-, Responsibility and Collaboration Cards) [4], [11], [12]. The notation used in the class diagram is illustrated in figure 10, which shows a class and four different structures. The notation is part of the Unified Modelling Language (UML), which has been chosen since it is the preferred standard worldwide and is used in many development tools [14].

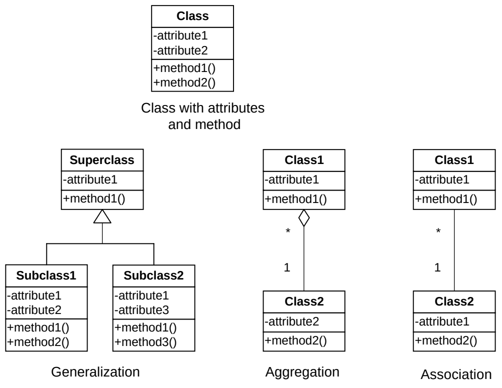


Figure 10. Notation for Class Diagram (UML) [4]

The second part of the OOA consists of an analysis of the field of application. Here the interaction between man and machine is analyzed in order to determine the functionality of the system, the user interface, and software integration to other IT-systems etc. Other elements that need to be determined are also requirements to response time, flexibility and so on. The result of this is a description of the user interface and a requirement specification for the product and product related models.

In order to model the switchboards at Demex an object-oriented model has been made. The model is based on the product master. It consists of a class diagram and CRC-cards. A part of the model is illustrated in Figure 11 and 12.

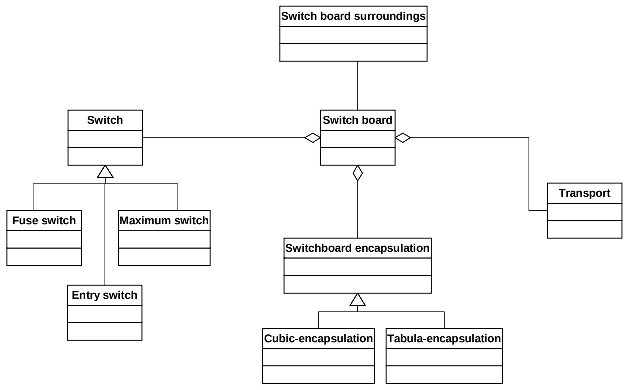


Figure 11. Class diagram.

Number: 18	**Name:** Entry switch
Date: 22.12.2000	**Version:** 2
Responsible person: JS	
Responsibilities: Specification of height and width of the encapsulation, BOM no., loss of heat, power and time for assembly.	
Attributes: Encapsulation [Tabula, Cubic] Dim [3x1, 2x1, 3x2] BOM no. [OT*] Price [DKr.] Time for assembly [min.]	
Sketch:	

Methods:

Name	Space (b x h)	BOM no.	Price [Kr]	Time for assembly [min]
OT 63	3 x 1	OT63T1	98,38	35
OT 63	2 x 1	OT64T1	132,86	35
OT 100	3 x 1	OT65T1	194,88	35
OT 100	2 x 1	OT66T1	229,36	35
QA 125	3 x 2	OT67T1	541,03	45
QA 125	3 x 2	OT68T1	580,33	45
QA 125	2 x 1	OT69T1	548,77	45
QA 200	3 x 2	OT610T1	562,66	45
QA200	2 x 1	OT611T1	554,50	45
QA 200	3 x 2	OT612T1	601,96	45

Collaboration:

With class 4, 8 and 11

Figure 12. Example of a CRC Card.

The software from Invensys CRM / Baan was selected for the programming phase.

3. 5 PHASE 5: PROGRAMMING

When programming a standard system the concepts for programming are sat by the supplier. The Baan Configurator is logic and constraint based [5], [6]. The product attributes and constraints are programmed based on the attributes and methods listed on the CRC-cards from the OOA-model.
The constraints are constructed with Boolean constraints, arithmetic constraints and warning constraints. Boolean constraints use logical operators like AND, OR, NOT, TRUE, FALSE etc. As an example a constraint concerning the Cubic module is showed below:

```
Cubic_module_CV AND
H_input_large_plus_Cubic_Y_DIN2M_CV3x2x1[0..2]
--> Cubic_CV[CV3x2x1]
```

The main part of the constraints in the system is of this type. Arithmetic constraints use operators like +, -, *, /, <, > etc. The last type is warning constraints that give a message if some criteria are broken.

There are approx. 12.000 Boolean variables and 7.000 constraints in the configurator, which is close to the maximum no. of variables and constraints the Baan Configurator can handle.

An example of a user interface is illustrated in figure 13.

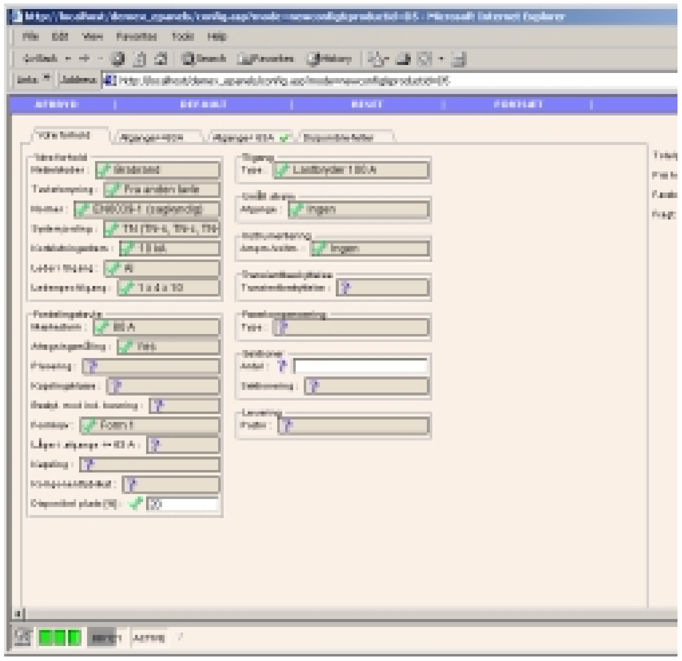


Figure 13. Example of user interface from the configurator.

4. CONCLUSION

The proposed procedure is based on well-known and proved theory elements. The aim of the procedure is to serve as guidance for engineers, working with product modelling. The procedure has been tested at several manufacturing companies in Denmark and abroad with positive results.

The proposed procedure includes several fields of expertise:

1. Business process reengineering, and business strategy
2. Product design and manufacturing technology
3. Theory for structuring mechanical systems, and structuring product- and product-related models
4. Object-oriented modelling
5. IT, Artificial intelligence and knowledge representation
6. Organizational aspects of application of product modelling

The wide range of theory is included in the procedure in order to cope with the questions raised in the introduction of the paper. I.e. how to deal with the business processes affected by the models, how to analyse and structure products and how to implement the models in IT-systems.

The project described in this article has shown how to build a configurator with automatic dimensioning and the specification of complex switchboards. The customers save time, money and energy when using the configurator. The users can now specify the demands to a switchboard system and then use the configurator as guidance when an optimal configuration is to be selected. Different parameters such as loss of heat and price summarisation of the system can be easily displayed. In this way corrections to a configuration will be shown immediately.

The effects of introducing a configurator in Demex electric/Solar A/S can be summarised as follows:

1. Reduction of lead time from 3-4 days to 10 minutes when generating quotes.
2. Possible to optimise a configuration in relation to e.g. resource consumption. This means up to 10 % reduction of materials.
3. Huge reduction in specification hours.

The configurator will be implemented in full scale at Solar A/S homepage in summer 2002.

6. REFERENCES

[1] Hvam, L., Riis, J., Malis, M. & Hansen, B., 2000, A procedure for building product models, Product Models 2000-SIG PM, Linköping, Sweden.

[2] Hubka, V. & Eder, W.E.: Theory of Technical Systems, Springer-Verlag. Berlin, 1988.

[3] Mortensen, N. H., Yu, B., Skovgaard, H. and Harlou, U.: Conceptual modeling of product families in configuration projects, 2000.

[4] Booch, G., Rumbaugh, J. & Jacobson, I., 1999, The Unified Modeling Language User Guide, Addison-Wesley.

[5] [Sabin et. al., 1998]: Daniel Sabin, Rainer Weigel; Product Configuration Frameworks – A Survey, University of New Hampshire and Swiss Institute of Technology, IEEE Intelligent systems, 1998.

[6] [Jackson, 1999]: Peter Jackson; Introduction to expert systems, third edition, Addison-Wesley, 1999.

[7] [Hammer, 1990]: Michael Hammmer; Re-engineering work: Don't automate, obliterate, Harvard Business Review, July-August, 1990.

[8] [Hammer et al., 1993]: Michael Hammer, James Champy; Reengineering the Corporation, Harper Collins Publishers, 1993.

[9] [Schwarze, 1996]: Stephan Schwarze; Configuration of Multiple-variant Products, BWI, Zürich, 1996.

[10] [Schwarze & Burke,1994]: Stephen Schwarze, L. Burke; The procedure of Product Configuration and Handling the Configuration Knowledge, Proceeding of the Third IERC, Atlanta, May 1994.

[11] [Tiihonen et. al., 1996]: Tiihonen, J., Soininen, T.,

Männistö, T., Sulonen, R.; State-of-the Practice in Product Configuration – a Survey of 10 Cases in the Finnish Industry, Helsinki University of Technology, 1996

[12] Hvam, L. & Riis, J.: CRC Cards for Product Modelling,

Department of Manufacturing Engineering, Technical University of Denmark, 1999. Antonio, Texas, November 17-20 1999.

[13] [Coad et al., 1991a]: Peter Coad, Edward Yourdon; Object-oriented analysis, Prentice-Hall, Inc., Second edition, 1991.

[14] Alexander Felfernig, Gerhard E. Friedrich And Dietmar Jannach; UML as domain specific language for the construction of knowledge-based configuration systems International Journal of Software Engineering and Knowledge Engineering, 2000.

Using Knowledge-Based Configuration for Configuring Software?

Lothar Hotz[1] and **Andreas Günter**[2]

Abstract. In this paper, we present a short survey on research topics which come into play when applying knowledge-based configuration techniques known in Artificial Intelligence (CAI) to the field of Software Configuration Management (SCM). Knowledge-based configuration deals with generic, logic-based methods for configuring components of a given domain. Typically these domains are hardware-based, like PCs, electrical drives, but in principle the methods are not restricted to those domains. Because the methods are well understood and configuration systems implementing those methods exist, it is natural to examine them in an upcoming project for configuring industrial products. Thus, this paper describes work that will be done based on work that was already done by us. As such, it is a position paper.

1 Introduction

Software is an important basis of most technical systems. Growing complexity and variability of technical systems replace the development of single software products with the development of product families. Furthermore, the increasing use of embedded systems combining hardware and software make the process of software development to a difficult task. To get started, we will analyse software product development processes of industrial companies, restricting us to industrial product lines with hardware and software components. As industrial products, we will focus on the car periphery supervision domain, which includes services like pre-crash detection, parking assistance, parking spot detection, blind-spot-supervision, and adaptive-cruise-control. Besides considering software and hardware components, also requirement templates, feature models, and intermediate representations will be examined.

To handle this task, configuration management systems have been developed. Properties of configurations, which are computed by such systems, should be:

- *consistency*, i.e. the components stacked together can be used together;
- *correctness*, i.e. not the wrong components (in respect to specific application needs) are selected;
- *completeness*, i.e. all necessary components are selected;
- *feasability*, i.e. a possibly good (not necessarily optimal) solution out of the big number of possible solutions is selected;
- *transparency*, i.e. controlling the process of configuration should be possible.

A prerequisite for such a support is a modeling method for the software to be constructed and a support for the configuration process. From the developers point of view, important modeling aspects are: modeling of evolution, concise representation of variability, optionality, dependencies/relations between components, non-complicated, non-chaotic modeling, architectural descriptions [3].

Knowledge-based configuration known in Artificial Intelligence provides a generic way for configuring technical artefacts. A model of a domain represented in a well-defined declarative configuration language is the basis of the configuration methodology. The configured results can be shown to have properties like correctness and completeness according to the specified model. Because of the generic approach, construction of software can be examined as a further application domain for CAI. In SCM a deep understanding of the domain of software configuration has been obtained, where in CAI, experiences in representing domain models and inferencing configurations from a model are available. Hence, a combination of both approaches might be promising [13].

This paper is structured as follows: First, we give a short presentation of basics of SCM methods and systems (Section 2) and a short overview of the methodologies developed in CAI (Section 3). In Section 4, we discuss the research challenges of SCM and their possible solutions in CAI. While these sections mainly deal with general aspects of combining CAI and SCM, in Section 5, we sketch a more restricted approach which we will follow in a new upcoming project.

2 Short view on SCM

Software Configuration Management systems focus on controlling the whole evolutionary process of software system development. This process is seen as a continuous process which does not stop while the software is in use. To support this constantly changing process, SCM systems have been developed. There are a number of SCM systems which support the main concepts of SCM systems more or less (for surveys in SCM see e.g. [2, 3, 6, 16]). These concepts are: The *representation* of software objects as *atoms* by giving them a rudimentary name and a not further specified content ”description” (e.g. ”program code”, ”documentation”, ”test result” etc.), or as *configurations* by enumerating independently changing atoms or other configurations. *Version control* examines how interim products are produced in the course of product development and how development of different aspects of the product can proceed in parallel. *Change control* examines how changes to the software objects are more or less formally described by giving information about the change like the objective of the change, the state of the change (e.g. open, rejected) etc. *Process control* supports the whole software development process, by describing the process in terms of ”completion, acceptance, integration & test, take over”. *Distribution* is related to distributed development of software by locally distributed developers.

[1] HITeC, Fachbereich Informatik, Universität Hamburg, email: hotz@informatik.uni-hamburg.de
[2] dito, email: guenter@informatik.uni-hamburg.de

Besides these basic concepts of SCM, there are further approaches especially in the research community in development but not included in commercial systems. The system Adele [4], (see also the discussion in [2]) is based on an entity-relationship database with more elaborate data modeling capabilities than those offered by file lists. However, drawbacks of such a system are: non strict object-orientedness, lack of sophisticated structured representations, incomplete support of attributes (the user has to manage some declared attributes instead of the system), no system independent semantics. Also the inference methods (which are based on so-called interfaces) are only centered around attributes and relations, not around software objects as a whole. However, with Adele it is possible to capture the evolution of all architectural elements in a single system model.

Summarising, conventional configuration management tools are intended to aid the production of single products, or products with limited variability. Although they offer some support for configuration, using them for families of products, which developers are currently forced to do, quickly leads to considerable adaptation effort through support utilities. Over and above this, there is little support for checking the consistency of configurations or for using inference to generate parts of the configuration.

3 Knowledge-based configuration known in Artificial Intelligence (CAI)

The configuration of technical systems is one of the most successful application areas of knowledge-based systems. [8] made a general analysis of configuration problems in which four central aspects (or knowledge types) concerned with configuration tasks were identified:

- A set of domain objects (*concepts*) in the application domain and their *properties* (parameters). By instantiating a domain object *concept instances* are created. Thus, a domain object describes its instances.
- A set of relations between domain objects. Taxonomical and compositional relations with alternatives, number restrictions, and optionality are of particular importance for configuration. But further relations can be expressed between arbitrary domain objects.
- A *task specification* (configuration objective) that specifies the demands, which a created configuration must fulfil.
- *Control knowledge* for specifying the configuration process.

For all those kinds of knowledge not only a model can be declaratively defined but also an inference machinery is given, which interprets the knowledge. Thus, CAI provides not only a modeling facility like STEP or UML but operational, processable models. In the so-called structure-based approach a compositional, hierarchical structure of the domain objects serves as a guideline for controlling the solution process, and as a logical basis for configuring. That means, that the constructs used for modeling can be mapped to a logical language where the semantics are well-defined [14]. The constraint-based approach consists of representing restrictions between objects or their properties by means of constraints, and evaluating these by constraint propagation. This approach is not in conflict with the structure-based approach but is frequently combined with it. Other approaches are resource-based and case-based configuration.

Concepts used in the area of configuration have well-defined, system independent semantics which are manifested in implementations termed *configuration systems* (CS) like EngCon [1]. Such systems provide a more formal notion of consistency and completion than Software Configuration Management systems [13]. For instance, in CS the consistency of the hierarchy is well-defined (namely a strict specialization hierarchy) and will be checked by a specific module of a CS. Constraint solving uses methods that have been proven correct, and property values of components can be inferred. The control mechanism determines that all open issues (e.g. properties, parts) of the configuration are handled by the configuration process [7]. All these modules of a CS are general and thus, domain independent. A domain specific configuration system can be obtained by implementing a domain specific application user interface over the domain specific configuration model. In CAI, this has been done traditionally for domains where hardware components are configured. For software, the main challenge is to understand the specific concepts of the software development process in terms of the general concepts of the logic-based configuration terminology.

4 Where are challenges/ problems in the field of SCM and their possible solutions with CAI

In this section, we discuss problems mentioned in the SCM community and their potential solutions with methodologies coming from CAI. We focus on *software product representation*, *versioning*, and *representing distinct kinds of knowledge*. Other aspects like evolution are presented in similar approaches like [12] and [5].

Software product representation In current commercial SCM systems *files* and file handling are the basic components, thus, operations like merging, revision etc. are based on files not on objects [3]. *Models* and software products (*instances*) are roughly seen as the same kind of entities namely software programs [5]. These facts demonstrate a main problem: there is no abstract, declarative model of the source code being configured [12]. Each task is done directly on source code and files. In CAI a configuration language is given which provides the means for defining the four central aspects of configuration tasks (see Section 3). With such a model, software products could be defined on a concept level, and the instances (e.g. files, source code), which realize a specific application, can be computed by the configuration system. An example for modeling a software system with CAI is given below. In this example, we describe parts of a tool box with the (here simplified) language used in EngCon. The example specifies the situation: ”`our-tool-box` is a kind of `software-system` where the parameters are specialized to the indicated, alternative values (between `{}`) and its diverse relations are specialized to diverse obligate or optional concept types (indicated by concept names and numbers)”:

```
def-concept
  :name our-tool-box
  :superconcept software-system
  :parameters
    system-purpose {configuration consistent-test layout-planning}
    procedure {not-fixed heuristic case-based}
    requirement-for-the-solution {no optimal}
    requirement-for-efficiency {no efficient very-efficient}
  :relations
    has-userinterface [userinterface 1 1]
    has-supporting-module
      [interface 1 1] [obk 1 1] [slot-contexts 0 1]
    has-basic-module
      [ontology 0 1] [constraints 0 1] [task-description 0 1] [control 0 1]
    has-extension-module [multiple-inheritance 0 1] [fuzzy-ontology 0 1]
      [fuzzy-constraints 0 1] [measurements 0 1] [requirement-modeling 0 1]
      [case-based-configuration 0 1] [optimal-configuration 0 1]
      [formulars 0 1] [taxonomical-inferencing 0 1] [backtracking 0 1]
```

An example for describing a relation between multiple components using constraints is given below. It is specified: ”An `our-tool-box` with an `interactive-strategy` as `extension-module` has to have exact one `basic-module` of the type `control` and exact one `userinterface` of type `graphical-userinterface`”. The ”?” is used to mask local variables:

```
def-conceptual-constraint
  :name interactive-strategy-requirements
  :selected-components
   ?system :name our-tool-box
   ?sub-module :name interactive-strategy
```

```
    :relations extension-module-of ?system
  :constraint-calls
    number-restriction
     (?system has-basic-module) [control 1 1]
    number-restriction
     (?system has-userinterface) [graphical-userinterface 1 1]
```

Versioning In [16] it is mentioned that versioning is mainly based on a directed acyclic graph by using the `is-version-of` relation. In CAI one could represent this aspect in the framework of specialization hierarchies. Here, versions can be described in terms of subconcepts with specializing properties or relations. Thus, also version management of structures, relationships and interfaces can be modeled, which is a further requirement mentioned in [6]. As described in Section 3, not only a model but also an inference machinery is given by CAI, hence, consistent configurations selected from multiple versions can be computed. Thus, the selection of appropriate components is dynamically supervised by the configuration system.

Representing distinct kinds of knowledge The CAI methodology is general and generic. Hence, distinct aspects like features, requirements, designs, test cases, task agenda, standard code artefacts and their relations, etc. could be represented (see [16]). These diverse aspects of a domain can be modeled in distinct knowledge basis and can be combined, e.g. by using *strategies* [7], in an integrated system. As an example, in [10] a feature model is described, which includes common and variable aspects of the capabilities of software products. Such a feature model could be used by a configuration system for identifying suitable software and hardware components.

There are some issues, which do not have an obvious solution and are still research aspects in both fields, CAI and SCM: combining CAI methods and SCM techniques, representing functionality of software modules [11], product variability [6], and distributed, concurrent work [6]. Aspects, which are more related to SCM, are discussed in [3, 6, 16]: Automated change integration and merging of changes; interoperability among configuration management systems; relationship between software architecture and configuration management systems; constructing executable object modules; representing data flow among modules; representing control flow among modules; deployment issues and post deployment phase; distributing software into the field and maintaining it. How these problems can be handled if CAI is used as a kernel technology, is still open.

5 Configuration of Industrial Product Families

In the new project *Configuration of Industrial Product Families (ConIPF)* following issues are examined:

- In order to support realistic industrial applications, guidelines will be described for facilitating domain modeling (see [13, 15] for similar approaches).
- A further focus is set to products that can be developed quickly and with little effort. On the one hand an early result can be produced and on the other hand, for those components libraries can be developed for reusing generic software components.
- For modeling, known configuration description languages will be used and possibly further developed for describing specific aspects of software modules, like functionality and state descriptions.
- A technology used for modeling will be Description Logics (DL), which provides a well-defined semantics for basic concept and role (=relation) definitions (for a description of a DL system see [9]). The combination of a DL with CAI will be examined to support inferencing on the concept level, for e.g. automatically classifing new component models in a specialization hierarchy.

The project is a three year EU-Project and is composed of two industrial partners: Robert Bosch GmbH, and Thales Nederland B.V. and two university partners: University of Groningen and University of Hamburg/HITeC.

6 Summary

SCM provides a clear view of the software construction domain, and of the main configuration aspects of that domain like evolution, versioning. CAI provides a generic kernel technology for configuring diverse types of knowledge. The logic-based representation and inferencing methods of CAI provide the basis for a new step in the direction of ”modeling instead of programming”, which is in our opinion the main challenge for software construction. By combining the already developed tools of SCM with the kernel technology of CAI, we see even further possibilities.

REFERENCES

[1] V. Arlt, A. Günter, O. Hollmann, T. Wagner, and L. Hotz, ‘Engcon - engineering & configuration’, in *Proc. of AAAI-99 Workshop on Configuration*, Orlando, Florida, (1999).

[2] S. Dart, ‘Concepts in configuration management systems’, in *Proc. of the 3rd. Intl. Workshop on Software Configuration Management*, Trondheim, Norway, (1991).

[3] J. Estublier, ‘Software configuration management: a roadmap’, in *ICSE - Future of SE Track*, pp. 279–289, (2000).

[4] J. Estublier and R. Casallas, ‘The Adele configuration manager’, in *Configuration Management*, ed., Walter Tichy, 99–133, John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, (1994).

[5] J. Estublier, J.M. Favre, and Morat P., ‘Toward PDM / SCM: integration?’, in *Proc. of the 8. Intl. Workshop on Software Configuration Management*, LNCS 1439, pp. 75–95, Bruxelles, Belgium, (July 1998). Springer Verlag.

[6] K. Frühauf and A. Zeller, ‘Software configuration management: State of the art, state of the practice’, in *9th International Symposium on System Configuration Management (SCM-9)*, Toulouse, France, (1999).

[7] A. Günter and R. Cunis, ‘Flexible control in expert systems for construction tasks’, *Journal Applied Intelligence*, **2(4)**, 369–385, (1992).

[8] A. Günter and C. Kühn, ‘Knowledge-based configuration - survey and future directions’, in *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, ed., F. Puppe, Springer Lecture Notes in Artificial Intelligence 1570, (1999).

[9] V. Haarslev and R. Möller, ‘Consistency testing: The race experience’, in *Proceedings TABLEAUX’2000*. Springer-Verlag, (2000).

[10] A. Hein, M. Schlick, and R. Vinga-Marting, ‘Applying feature models in industrial settings’, in *Software product lines - Experience and research directions*, ed., Donohoe P., pp. 47–70. Kluwer Academic Publishers, (2000).

[11] C. Kühn, ‘Modeling structure and behaviour for knowledge based software configuration’, in *14th Workshop, New Results in Planning, Scheduling and Design (PuK2000)*, ed., http://www-is.informatik.uni oldenburg.de/sauer/puk2000/paper.html, (2000).

[12] T. Männistö, *Towards Management of Evolution in Product Configuration Data Models*, Ph.D. dissertation, University Helsinki, 1998.

[13] T. Männistö, T. Soininen, and R. Sulonen, ‘Product configuration view to software product families’, in *Software Configuration Workshop (SCM-19)*, Toronto, Canada, (2001).

[14] R. Möller, C. Schröder, and C. Lutz, ‘Analyzing configuration systems with description logics: A case study’, in *http://kogs-www.informatik.uni-hamburg.de/~moeller/publications*, (1997).

[15] J. Tiihonen, T. Lehtonen, T. Soininen, A. Pulkkinen, R. Sulonen, and A. Riitahuhta, ‘Modelling configurable product families’, in *Proc. of the 4th WDK Workshop on Product Structuring*, Delft, The Netherlands, (October 1998).

[16] A. van der Hoek, D. Heimbigner, and L.W. Wolf, ‘Does configuration management research have a future?’, in *Proceedings of the 5th Inter. Conf. on Software Configuration Management, LNCS 1005*, Berlin, (1995). Springer-Verlag.

Experiences with a procedure for modeling product knowledge and building product configurators - at an American manufacturer of air conditioning equipment

Benjamin Hansen[1] & Lars Hvam

Abstract

This paper presents experiences with a procedure for building configurators. The procedure has been used in an American company producing custom-made precision air conditioning equipment. The paper describes experiences with the use of the procedure and experiences with the project in general.

Introduction

At the Department of Manufacturing Engineering and Management at the Technical University of Denmark there have been several research projects involving the application of expert systems/configurators in manufacturing and engineering companies. Configurators have been built to support the creation of customer specific quotes, bill-of-materials, drawings, diagrams, routings, etc, i.e. specifications created in the customer-related business processes.

Since most mass-producing companies do not really have complex configuration tasks in the customer-related processes, most research projects have been carried out in engineering companies. These companies are typically small and medium-sized companies that customize products to fit specific customer needs. Often these companies have not been highly industrialized because it has not been possible to automate knowledge work. All over the western world such companies exist and they need to cut costs and reduce lead times to survive in the global economy. Different IT systems such as CAD, PDM and configurators may help to reach this goal.

This paper presents experiences with the use of the DTU [Hvam et. al, 2000] procedure during a configuration project in such a company.

The case company

The case company is an American company with 200 employees, manufacturing custom-made precision air conditioning equipment. The company was recently acquired by a much larger company and is now a division in that company. The customers are primarily companies in need of precision cooling to maintain the right inner climate in rooms with electronic equipment.

The objective of the acquisition is to transform the former small engineering company into a highly modern division that can mass-customize products on a large scale. An important part of this is to automate the customer related configuration process with IT. This led to an initiative of building a configurator for the most suited air product family.

Figure 1: A picture of the air configuration equipment

The project

The project followed the procedure used in projects connected to research at DTU. The project procedure is illustrated in Figure 2. The main experiences with the use of the procedure are discussed for each phase.

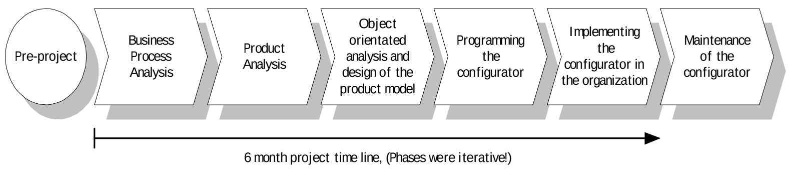

Pre-project
Business Process Analysis
Product Analysis
Object orientated analysis and design of the product model
Programming the configurator
Implementing the configurator in the organization
Maintenance of the configurator
6 month project time line, (Phases were iterative!)

Figure 2: The general procedure with indication of specific project activities.

Pre-project initiatives

Before the actual project started there had been a rough analysis of the business process and the product to be configured in it. This analysis indicated that an IT-based configuration of a quote and of a bill-of-material was possible, but that it a simplification of the product structure was necessary to be able to do it efficiently. The main arguments for doing so were the following:

- There were too many low level part numbers being involved in the configuration process.

[1] Ph.D. student at Department of Manufacturing Engineering and Management, Technical University of Denmark. e-mail: blh@ipl.dtu.dk

- The mother company requires all configure-to-order products to be structured in subassemblies and designed for easy assembly.

This resulted in a new assembly structure of the product in manufacturing. All low-level components were grouped into sub-assemblies that were classified in module classes. The final product now consists of a model with 12 classes, with 1 or more sub-assemblies/modules in each class. Each sub-assembly consists of 1-50 low-level part numbers. Thus the assemble-to-order model is no more than 3 levels deep, consisting of a top level product variant, a list of sub-assemblies and a list of parts in each sub-assembly.

As stated often under Business Process Re-engineering projects it is important to simplify a process before automating it [Hammer, 1990]. This was exactly what was done through the modularization efforts [O'Grady 1999]. It made the configuration task a whole lot simpler, which meant that the basic structure of the configuration process was simplified dramatically. This simplification of the product structure was not an easy task. In fact it was not at all finished when the configuration project was started, which meant that rules where changing during the modeling phase, and that a lot of time was spent waiting for updated lists of the assembly "super" BOM.

Thus, one of the main experiences from this project is:

- Redesigning the product structure and defining sub-assemblies was the biggest single task related to the configurator project.

Business Process Analysis

The business process analysis is made to get an overview of the specifications created in the process, and to define what the requirements are for the configurator to be built. To support this the procedure presents some analysis tools (IDEF0, flowcharts, and a list of characteristics to be analyzed to understand the task of the engineering/configuration process).

Through interviews with the people involved a flowchart of the process was drawn, and examples of specifications created in the process were collected. This gave an impression of the flow and the information/specifications in the process.

Some typical characteristics of configuration processes were analyzed as illustrated in table 1.

Characteristics	**Analysis result**	**Future requirement**
Input/Output	The customer specifies a *list of 20-40 standard features*. Based upon this a *priced quote* is given. Then a detailed *proposal/submittal* is made. Internally the specifications: *manufacturing BOM, arrangement diagram, drawings, electrical wiring* and a *shipping BOM* are made.	The configurator must generate 80% of *quotes, manufacturing BOM's* and *shipping BOM's*.
Frequency	10-15 orders are processes each week. The number of quotes is 3-5 times higher.	The configurator must be able to support a frequency 10 times higher than today.
Throughput time	"clean orders" ship within 5 weeks. 2-6 weeks are used during specification activities. Few orders are "clean orders"	The market requires a significant reduction of lead times.
Resource consumption	1-3 hours are spent on each quote. 1-3 hours on BOM creation.	Resource consumption should be decreased, but is not critical.
Quality	The quality of specifications is poor. A very manual process with a lot of paper work result in misunderstandings. This may led to wrong items being shipped.	Quality must be improved.
Availability	The availability of specifications is poor. A high degree of paper makes it difficult to access documents and information when it is needed	Quotes must be made available to all sales partners on the internet.
Learning curves	The learning curves are 1-2 years. This causes big problems when new people are assigned.	Knowledge must be put in a configurator.

Table 1: An analysis of specific characteristics of the business process

The main experiences with the business analysis were:

- The analysis tools helped but,
- Many decisions were actually based on "guts-feelings" from top management.
- The learning process was seen as more important than short-term economical cost/benefit evaluations.

Product Analysis

The structuring matrix [Andreasen, 1996] and the principles of building a product variant master [Mortensen, 2000] were used to analyze the product. These tools are presented in the procedure as a means to analyzing the product.

A very important aspect that came to light during the analysis was the fact that the air-conditioning product did in fact have different structures in sales, manufacturing and shipping (also illustrated in Figure 4). This was hard for some top-level managers to accept, they kept demanding one single structure. These three viewing angles did have to be modeled separately, each getting their own superstructure model (Product Variant Master), just as the specifications generated from the configurator would be different for sales, manufacturing and shipping. This separation of the product model into 3 sub-models made it a lot simpler to model the knowledge. The product analysis resulted in 3 rough sketches of "product variant masters ".

The most important experiences were:

- It was very helpful to have a clear understanding of the product before the actual design of the detailed product model was carried out. This made it possible to make a better overall structure of the product model. It also supported the planning of the knowledge acquisition process needed for the detailed object-oriented design of the product knowledge (next phase).
- The product was seen from very different angles in sales and manufacturing. It was hard for some people to understand that a product may actually have different structures depending on the point of view.
- Using the theory of "product structuring" [Andreasen, 1996] helped people without product design knowledge to understand the product better.

Object-oriented analysis and design of the product model

Based upon the analysis of the business process and the product, a model of the acquired knowledge was built. Principles from object-oriented design were used to model the knowledge [Booch et. al 1999]. These principles are basically to structure the knowledge in classes following an aggregation structure similar to the products "super-BOM", and to attach attributes and rules to each class. This encapsulates the knowledge in objects that are easy to maintain and reuse. The actual tools used were so-called "Class-Responsibility-Collaboration" cards (CRC-cards) [Hvam&Riis, 1999], which are used to model attributes and methods for product models. Positive experiences at another company [Hvam&Malis, 2001] with Lotus Notes as a document database, led to the design of a specific Lotus Notes database to handle the knowledge acquisition process Figure 3.

The Lotus Notes database with CRC-card documents now serves as a link between the domain expert and the programmer. Here there is a written "agreement" on all the knowledge used in the configurator. A rule must be documented and maintained here before it is modeled in the actual configurator. When in doubt about what rules were given from the domain expert to the programmer, these CRC-cards served as written proofs of what was agreed upon. Using Lotus Notes makes it possible to work together over long distances, which is everyday life in the company.

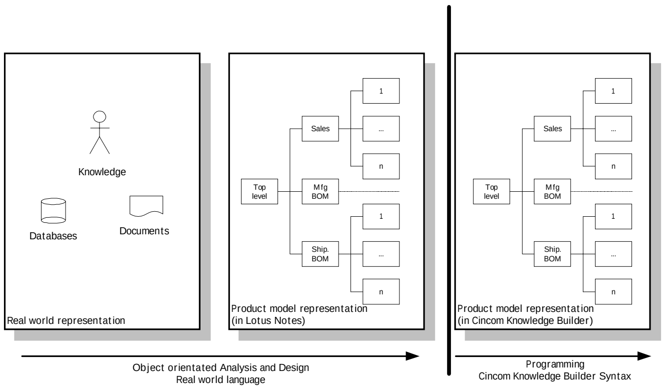

Knowledge
Databases
Documents
Real world representation
Top level
Sales
Mfg BOM
Ship. BOM
1
...
n
Product model representation (in Lotus Notes)
Product model representation (in Cincom Knowledge Builder)
Object orientated Analysis and Design
Real world language
Programming
Cincom Knowledge Builder Syntax

Figure 3: The modeling process

The model in Lotus Notes is grouped into 3 sub-models: a feature model used in sales, an assemble-to-order model used in manufacturing, and finally a shipping model.

The main experiences with this phase were:

- Object-orientated modeling and the CRC-cards made it easier to structure the knowledge. Exact syntax and semantics as used in software development were not used, since this was too complex for the domain experts.
- Data redundancy occurred: The knowledge documented in the model did refer to data and knowledge used and stored in other systems in the organization. Part numbers (sub-assemblies) were documented in a central part specification database, and in the ERP system. Now they had to be documented in the knowledge model as well (and later in the actual configurator software). This data redundancy was a problem that was well understood, and it led to an integration of integrating the CRC-cards in the Lotus Notes database with the corporate part spec. database (which was also a Lotus Notes database). Regarding the knowledge there were as many levels of redundancy as people in the organization. Having an easily accessible knowledge representation on documents in Lotus Notes made it possible to reduce this knowledge

redundancy. In principle there would be no redundancy since there is only one documented knowledge model. In reality it has been difficult to make people understand the importance of updating knowledge in this single repository, still making errors of not having the latest knowledge in the database possible.

- Using Lotus Notes to store the CRC-cards with attributes and rules, made it a lot easier to collect and document knowledge.

Programming in Cincom Knowledge Builder
Cincom's configurator software called Knowledge Builder (KB2K) [www.cincom.com] was used as configurator software. This software has been used with success in the mother company for several of the projects and therefore it was decided to use this tool again.

The structure of the product model in the program is very similar to the structure of the product model in Lotus Notes. For each CRC-card there is a corresponding group of programming code in KB2K. Part of the code is directly representing the product model, while another part represents system specific code used for UI navigation, web integration etc. The model part of the code is structured into three main categories: sales, mfg. BOM, and Shipping BOM Figure 3.

To ease maintenance it has been decided to follow a strict pattern using Boolean constraints to validate the selection of features (representing the sales view on the product) on the user interface. After the features have been selected, IF-THEN rules are used to pick the right BOM-numbers on the respective "Super BOM structures" for assembly and shipping. This is illustrated in Figure 4 .

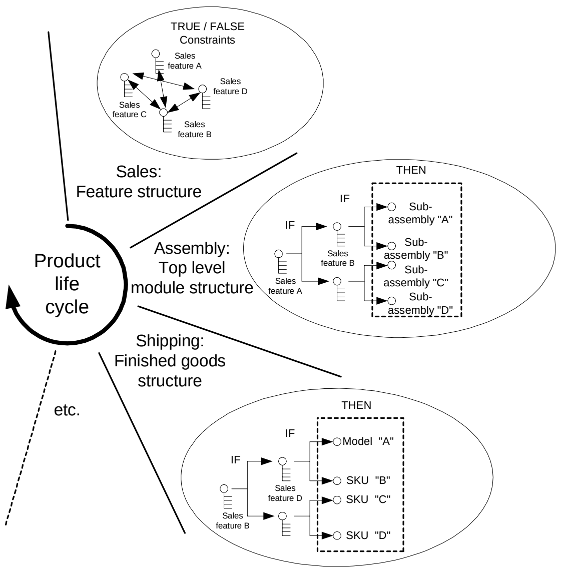
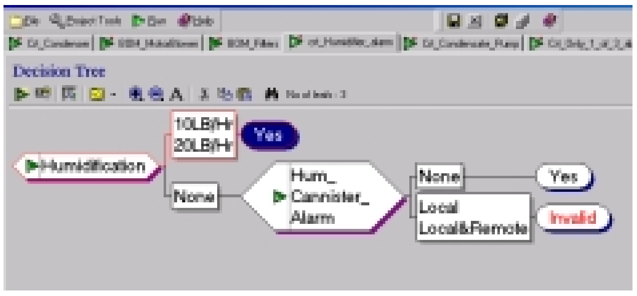


Figure 4: How constraints and IF-THEN rules are used in the 3 different parts of the configurator

The separation between sales features (represented on the UI) and BOM numbers makes it easier to maintain the configurator. BOM numbers and their selection rules can be modified and maintained in one part of the code, while UI, sales features and the constraining (Figure 5) of these can be maintained in other separated parts of the code.



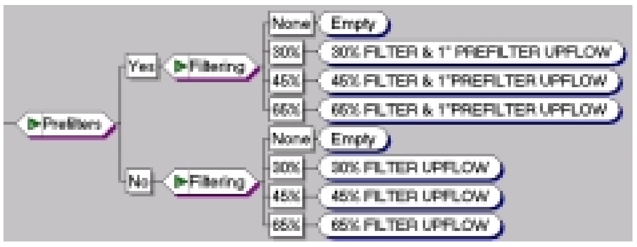


Figure 5: Constraints (top) and rules (buttom) in Cincom Knowledgebuilder

Generating output data is done after all features are selected. When pressing "done" on the last feature selection page, the configurator starts running over all rule-trees on the superstructures (Figure 5). This results in several data-strings, representing the configuration in a sales view (quote), a manufacturing view (manufacturing BOM) and shipping view (Manufactured unit + BOM for shipped standard items). These data strings are then exported to a database, from where the data can be used in various applications. In this case they are used for generating 3 types of reports: a quote, a manufacturing BOM, and a shipping BOM.

The main experiences were:

- Having a product model on "paper" (Lotus Notes) made the programming a whole lot faster. Having data and knowledge written down in the Notes database made it possible to utilize programming resources better.
- Without giving a detailed evaluation of Cincom's Knowledge Builder it can be concluded that the experiences with this tool were all positive. Especially the flexibility of the tool to represent different kinds of knowledge in different ways was helpful. When standard-modeling techniques were not enough, a high-level programming language could be used which also gave a great flexibility.
- It was possible to represent the programming in Cincom in the same object-oriented aggregation structure as in the product model (using "Categories"), which made maintenance much easier.

Implementation and description of the solution
The final solution is very similar to other configurators generating a quote and a BOM. It runs on the web on an

intranet server. Data are stored in a central SQL database. Currently the configurator is mainly used internally to validate quotes and to generate BOM's. It does fulfill the requirements set up for it during the business process analysis.

The most important experiences with the implementation were:

- It took more time than expected to implement the configurator in the organisation. Several details and requirements were not defined until the users actually experienced the configurator through a first hand experience.
- Due to the object oriented product model which was well documented in Lotus Notes and due to a similar object oriented modeling in Cincom it was relatively easy to make the necessary changes in the configurator during the implementation.

Conclusion

Using the procedure helped to structure the project better. The specific tools presented in the procedure helped in the different activities where they were applied. But a well-structured project framework with specific tools did not make up the success on it's own. In fact other factors were even more important:

- The strategic focus on mass customization in the mother company and a vision to support it ensured the necessary commitment and drive from the people involved in the project. Without this it would have been much harder to get through with the project
- Experiences with other configurator projects did exist. It was well known that the product would have to be modularized, and that people would have to be motivated in order for the project to become successful. Furthermore Cincom had been used in the mother company for several projects, which meant that there were "experts" available to guide through technical issues. Thus experiences with this type of projects are just as important as a structured procedure.
- A network to other companies doing configurator projects was more or less established before the project started. This meant that some typical problems could be avoided, and that inspiration for specific solutions could be found.

The case shows that transforming specification processes from manual engineering processes to automated configuration processes is a very complex task. The IT aspect is only a smaller part of the development process, even when the goal is to build an IT system. The company must have a good understanding of areas such as product structuring, knowledge acquisition, business process reengineering, project management, configuration and database technology etc.

The mother company that bought the little engineering company was aware of this complexity. Therefore this configuration project, the first of many has been regarded as a learning project. Presenting a clear vision, motivating people, and being willing to spend a lot of resources on learning by doing, is believed to make it possible for the organization to cope better with the complexity of the transition towards mass-customization.

References

1: [Andreassen, 1996] Andreasen M.M. , Hansen C.T. , N.H. Mortensen: "The structuring of products and product programmes", Proceedings of the 2nd workshop on product structuring, 3-4th of June 1996, Delft University of Technology 1996.
2: [Booch et. al 1999] Booch G., Rumbaugh J. & Jacobsen I.:"The Unified Modeling Language User Guide", Addison-Wesley, 1999.
3: [Hammer, 1990] Hammer M. "Re-engineering work: don't automate, obliterate", Harvard Business Review, July-August 1990.
4: [Hvam et. al 2000] Hvam L. , Riis J., Hansen B. , Malis M. "A procedure for building product models", Proceedings from the conference: "Product models 2000", Linkoeping, Sweden, 7-8th November 2000.
5: [Hvam & Riis, 1999] Hvam L. Riis J. "CRC cards for product modeling", The 4th Annual International Conference on Industrial Engineering Theory, San Antonio, Texas, November 17-20, 1999.
6: [Hvam&Malis, 2001] Hvam L. , Malis M. , "A knowledge based documentation tool for configuration projects", World Congress on mass customization and personalization, October 1-2, Hong Kong 2001
7: [Mortensen 2000] Mortensen, Yu, Skovgaard and Harlou: "Conceptual modeling of product families in configuration projects", 2000.
8: [O'Grady, 1999] O'Grady Peter. "The age of modularity", Adams and Steele Publishers, October 1999.

Fuzzy Case Based Configuration

Laurent Geneste[1] and **Magali Ruet**[1]

Abstract. We propose in this paper to define a configuration process based on past configuration experiences. Similar configuration problems are in this framework expected to have similar configuration solutions. An integration of Case Based Reasoning and Constraint Satisfaction techniques (CSP) to support the configuration process is suggested. Since the manipulated information is complex and imprecise, we propose to use a fuzzy object oriented representation and to define appropriate algorithms for CBR and CSP integration. We illustrate our proposition by an example about the configuration of a machining operation.

Keywords. Configuration reuse, fuzzy knowledge base, FCSP, CBR

1 INTRODUCTION

Many configuration methods propose to use CSP techniques in order to deal with the complexity of configuration problems. These techniques do not take into account past configuration problems that may guide a new configuration. Our aim is to propose a configuration process taking into account past configuration experiences. We suggest to search the solution of a configuration problem in the neighbourhood of a similar past configuration. That is why the use of case based reasoning (CBR) paradigm is proposed. Moreover, in order to support the reuse and adaptation of a relevant past configuration, constraint satisfaction techniques are integrated in the CBR process.

An object oriented modelling is used for representing configuration cases. A case is modelled by an object composed of characteristics (attributes). An attribute can be described by another object (composition). The possibility to manipulate imprecise characteristics of cases is integrated. Indeed, the imprecision on the values of the characteristics is represented by possibility distributions [5]. A characteristic A is defined on a reference domain R by a possibility distribution which express the membership of each value of R to the characteristic A. A possibility distribution is a function π of a reference domain R to [0,1] such that $\sup \pi(x) = 1, x \in R$.

In this paper we propose to develop the way we reuse past configuration experiences. In section 2, we first shortly describe how the search process of past configuration experiences is carried out. Then we propose, in section 3, to use techniques of CSP for the adaptation of past configuration experience. Finally, we illustrate our proposal on an example about the configuration of a machining operation.

2 CASE BASED REASONING FOR CONFIGURATION PROBLEM

Case based reasoning is a paradigm that proposes to use past experiences in order to solve a current problem, when "similar problems have similar solutions" [11]. In the CBR process, a past experience (source case) similar to the current problem (reference case) is retrieved and its solution is adapted to the current problem.

Different applications propose to use Case Based Reasoning techniques for configuration. For instance, in order to configure a Personal Computer, past pre-configured PC cases are reused and adapted according to user requirements [1]. Another example concerns the elaboration of personal Electronic TV Programme Guide (EPG): past similar programmes chosen by a user are reused and combined with items recommended by similar users for configuring a personal user guide [3].

In order to solve a problem, several steps compose the Case Based Reasoning process:

- model the problem,
- retrieve past problems,
- reuse the most similar past problem by adapting its solution,
- revise the proposed solution,
- and eventually retain this new case (the problem and its solution) for future use.

All along this process, the user can intervene in order to guide the process and to control it.

We introduce in section 2.1 mechanisms we use in order to retrieve past configuration problem. Then in section 3 we present our work in combining CSP techniques with the adaptation step of the CBR process for configuration.

2.1 Search of past similar configuration

The aim of the search step is to provide the past configuration experience which is most similar to the current problem. A similarity measure is applied between each possibly similar source cases (resulting from a rough filtering process which is not developed here) and the target problem. This similarity measure takes into account the object oriented structure of cases and enables the use of possibility distribution in order to represent the imprecision on the values of characteristics of cases [9]. The result of the similarity measure is divided into two degrees of similarity: N is the degree of necessity of resemblance of two cases and Π is the degre of possibility of resemblance of these cases (N and Π are in [0,1]).

We distinguish the local similarity which is computed at attributes (characteristics) level and the global similarity which is an aggregation of local similarities and is computed at object level.

The local similarity is computed thanks to a similarity membership function that enables a user to associate to an attribute a specific way to compute the similarity (e.g. the membership function *near to,* defined by $\mu_L(x,y) = 1\text{-}|x\text{-}y|/\Delta$ where Δ is a constant). The user also controls the global similarity (at object level) by weighting each attribute in the aggregation.

The similarity measure is computed as fallow:

Notations:

- R denotes the reference case and S the source case
- $att_{R,L}$ the name of attribute L of case R; $val_{R,L}$ its value
- D_L the domain of attribute L and $U = D_L \times D_L$
- w_L the weight associated to attribute L for the search
- μ_L the membership function describing the local similarity for attribute L
- π_R the possibility distribution describing $val_{R,L}$
- π_S the possibility distribution describing $val_{S,L}$
- π_D the possibility distribution defined by

[1] Equipe PA - LGP - ENIT - Avenue Azereix, 65000 Tarbes, France, email: laurent@enit.fr, ruet@enit.fr

$$\pi_D(x,y) = \min(\pi_R(x),\pi_S(y)) \quad (1)$$

At the level of each attribute L, the possibility and necessity degrees corresponding to a local similarity are computed as follows:

$$\Pi_L(val_{R,L},val_{S,L}) = \sup_{u\in U} \min(\mu_L(u),\pi_D(u)) \quad (2)$$
$$N_L(val_{R,L},val_{S,L}) = \inf_{u\in U} \max(\mu_L(u), 1-\pi_D(u))$$

The necessity and possibility degrees represent to which level two cases are similar. They respectively correspond to the lower and upper bound of the similarity degree.

After the evaluation of each local similarity the evaluation of the global similarity is achieved, taking into account the weights associated to each characteristic of the target problem:

$$\Pi(R,S) = \min_{i=1,n} \max(1-w_i,s_i) \quad (3)$$
$$N(R,S) = \min_{i=1,n} \max(1-w_i,s_i')$$

with :

$$s_i = \begin{cases} \Pi_L(val_{R,i}, val_{S,j}) & \text{if} \quad \exists j \in \{1,\dots,n\} att_{R,i} = att_{S,j} \\ 0 & \text{else} \end{cases}$$

$$s'_i = \begin{cases} N_L(val_{R,i}, val_{S,j}) & \text{if} \quad \exists j \in \{1,\dots,n\} att_{R,i} = att_{S,j} \\ 0 & \text{else} \end{cases}$$

In addition to the similarity measure we use an adaptability measure that reflects how easy past experiences can be reused. See [10] for more details.

The result of both similarity and adaptability measures allow to select an appropriate past configuration experience to continue the process. The solution of this past experience has to be adapted. We propose to use constraint satisfaction techniques during this adaptation process. We develop this point in the next section after a brief literature overview.

3 INTEGRATION OF CBR AND CSP TECHNIQUES FOR CONFIGURATION

A lot of works propose to combine CSP and CBR techniques as stated in [19]. We are mainly interested in works where constraint satisfaction techniques and configuration techniques are used in the adaptation process of case based reasoning paradigm. We describe these works in section 3.1 before presenting our proposition in section 3.2.

3.1 Related works

In the COMPOSER system [16] CSP techniques are used to support the adaptation process of CBR in the field of engineering design. The proposed methodology uses cases represented as a discrete CSP. A matching process is applied between old cases and the new problem; several cases emerge of this matching. These cases, their solutions and their constraints form a new problem: the new CSP which can be solved with CSP algorithms. In COMPOSER all cases must be modelled as a CSP in order to apply constraint satisfaction algorithm to adapt found cases.

In [14] the authors propose to use the CBR paradigm on constraint satisfaction problems in product configuration. The case based reasoning process is used to help the customer in the expression of his needs. Past cases represent past sales and describe past buyers and the product they bought. The adaptation process is achieved thanks to interchangeability [15] [21]. In fact, in a CSP, in some situations, a value can be replaced by another value. Here the problem is to know which variables can be replaced by which other variable(s), and to use this knowledge in order to solve the CSP. This is what is done in the CBR adaptation process.

The IDIOM system [12] aims at promoting interactive design of building by reusing past designs and adapting them according to preferences on the design and on the combination of cases. Past design cases are stored and when an architect designs a new building, he selects past cases and may add preferences on them if necessary. Along with past cases, constraint on cases combination are also stored. The past cases are combined to form a new design. Cases are then adapted with CSP algorithm according to knowledge they hold and to preferences.

The approach defined in [20] for testing the interoperability of networking protocols suggests to represent the knowledge base (Interoperability tests) as a set of CSPs. When a CSP fails according to monitored observations, CBR is used to enhance the CSP according to previous similar cases.

In the field of configuration, the authors of [21] introduce the idea of starting from a previous configuration close to the customer requirements (instead of starting from scratch) and to adapt it. In this system, cases are modelled according to the CSP representation and adapted thanks to neighbourhood, context dependent and meta interchangeability concepts

All these works are based on a CSP model of cases. This modelling seems to be not sufficient to represent expert knowledge. We propose to use an object oriented modelling of cases which allows to model expert and complex knowledge. Moreover the model we use permits to represent constraints between objects. Hence, cases are not constructed in the form of a problem of CSP.

In next section, we develop the way we integrate CSP techniques in a CBR process for configuration.

3.2 CSP for CBR adaptation

CSP techniques can be used for guiding the adaptation of the past solution to the current problem. In fact, when a case is retrieved in the case base, we propose to find a solution in the neighbourhood of this case by constraining this neighbourhood. We need first to define this area called adaptation domain. We expose our method to determine it in section 3.2.1. Once the adaptation domains are found, the propagation of the constraints can begin. We describe our proposition in section 3.2.2.

3.2.1 Adaptation domain

In order to determine an area all around the retrieved case, we propose to calculate adaptation domain with three parameters: the values of the retrieved case (values of its attributes), the similarity membership function of each attributes of the retrieved case and a value α that represent a similarity threshold.

The similarity membership function of an attribute (μ_L) and the value of this attribute (μ_S) are used to determine a new membership function (μ_R). This one is calculated by the projection on X axis of the intersection of μ_L and μ_S (Fig 1). Then the α-cut is computed in the new membership function μ_R (Fig 2). The result of this calculation gives the adaptation domain of the attribute.

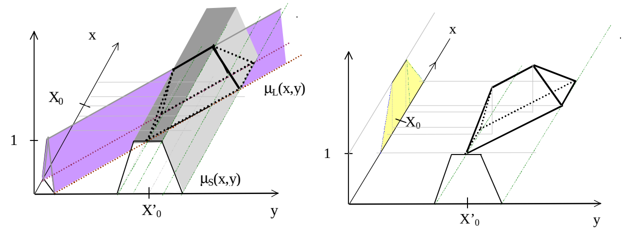


Figure 1. Intersection between μ_S(x, y) and μ_L(x, y)

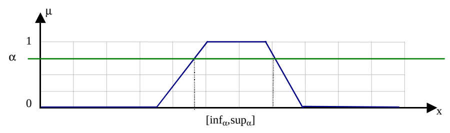


Figure 2. α-cut of the membership function $\mu_R(x,y)$

3.2.2 Fuzzy constraint propagation

In order to propose a good solution to the user, we suggest to use CSP techniques to guide the adaptation process. We propose to propagate domain constraints on the adaptation domain previously determined.

The various domains of the constraints (discrete, continuous, both), the arity of the constraints (binary, n-ary), and the dynamic of the constraint application enable to select between several propagation techniques such as [2], [4], [8], [18] as suggested in [13].

But we expect to use more of the CSP techniques possibilities. In fact, we propose to use fuzzy CSP as described in [6]. First, the use of soft constraints is in accordance with our proposition that allows to model imprecise and uncertain knowledge and data. Second, prioritized constraints and prioritized soft constraints allow to personalize the constraints description.

Before developing our proposition, we first remind notions on soft constraint.

As defined in [6] a soft constraint C is described by a fuzzy relation R so that:

Let D_1 x…x D_n be a fuzzy set of values that more or less satisfy C ($D_1,\ldots,D_n$ are the respective domains of the variables $\{x_1,\ldots,x_n\}$ of the constraint C),

then the fuzzy relation R is defined by a membership function μ_R which associates to each tuple $(d_1,\ldots,d_n) \in D = D_1$ x…x D_n a degree of satisfaction $\mu_R(d_1,\ldots,d_n)$ in [0,1]. This degree expresses to what level the solution $d = (d_1,\ldots,d_n)$ is compatible with the constraint C:

$\mu_R(d_1,\ldots,d_n) = 1$	means that $(d_1,\ldots,d_n)$ totally satisfies C
$\mu_R(d_1,\ldots,d_n) = 0$	means that $(d_1,\ldots,d_n)$ totally violates C
$0<\mu_R(d_1,\ldots,d_n)<1$	means that $(d_1,\ldots,d_n)$ partially satisfies C

Hence, a soft constraint expresses preferences among solutions. A solution satisfies a constraint with a degree of satisfaction. Hard constraint are particular soft constraint which degree of satisfaction is 1 or 0 only.

The author of [7] proposes an extension of AC3 filtering algorithm integrating fuzzy constraints called FAC-3 defined as follow:

Let P be a fuzzy CSP defined by : P = (X, D, C, R), so that:

- X is the set of variables
- D is the set of domains of the variables
- C is the set of constraints
- R is the set of fuzzy relations defining each constraint. R_i is defined by a fuzzy set. R_i is the set of values that satisfy more or less the constraint C_i.

Let Cons-P-sup be the upper approximation of the overall consistency degree, and V(R) be the set of variables related by R, then:

```
**Procedure FAC-3** (**P** = (X, D, C, R), $\beta = 0$)
Cons-P-sup $\leftarrow$ 1
Q $\leftarrow$ { (i,j) / $\exists$ $C_h \in$ C s.t. V($C_h$) = {$X_i$, $X_j$}, i $\neq$ j }
while Q not empty and Cons-P-sup > $\beta$, do
    select and delete (i,j) from Q
      if Revise (i, j, Cons-P-sup) do
          Q $\leftarrow$Q $\cup${(k,i) / $\exists$ $C_h\in$ C s.t. V($C_h$)={$X_i$, $X_k$},k$\neq$i,k$\neq$j}
return Cons-P-sup.
```

```
**Procedure Revise (i, j, Cons-P-sup)**
Changed $\leftarrow$ false
Height $\leftarrow$ 0
for each $d_i$ of Support($R_i$) do
    cons $\leftarrow$ 0
    for each $d_j$ of Support($R_j$) do
        cons $\leftarrow$ max ( cons, min ( $\mu_{Ri}(d_i)$, $\mu_{Rij}(d_i,d_j)$, $\mu_{Rj}(d_j)$))
    Height $\leftarrow$ max (cons, Height)
    if cons $\neq$ $\mu_{Ri}(d_i)$, do
        Changed $\leftarrow$ true
        $\mu_{Ri}(d_i)$ $\leftarrow$ cons.
Cons-P-sup $\leftarrow$ min (Cons-P-sup, Height)
return Changed
```

Based on this filtering algorithm and associated with a search algorithm such as Branch and Bound, we can propagate domain constraints on the adaptation domains [6]. In the next section we propose an example showing such use of CSP algorithms.

4 EXAMPLE

We illustrate our propositions by an example on the configuration of a machining operation. A machining operation is made of a part, a tool and a machine. In this exemple, some attributes are described thanks to fuzzy number (Fig 3). A machining operation is represented by an object diagram (Fig 4).

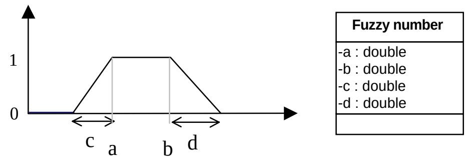


Figure 3. Fuzzy class model

An instance of the class diagram is described in figure 6, in which we can see four different operations recorded in the knowledge base (please note that to be make the schema easier to read, attributes are written with abbreviations detailed in figure 3). The operation to configure, called OpX, is represented in figure 5. This target operation has some attributes known and others that are not valued. In this example we present the search step of similar operation (section 4.1) and the adaptation process (section 4.2).

4.1 Search of the most similar operation

Based on the four operations of the knowledge base and on the works previously presented, we can describe the search of the most similar and adaptable past configuration.

In order to compute the similarity and adaptability measures of each past cases we have to weight attributes of OpX and to select similarity membership functions which have to be used.

In this example, importance is given to attributes "cutting direction" and "relief angle", attributes of the tool: their weight are of 1. Numbers placed in front of the characteristics in figure 5 correspond to the weight of characteristics of the operation to configure.

The similarity function membership used in this example are as fallow:

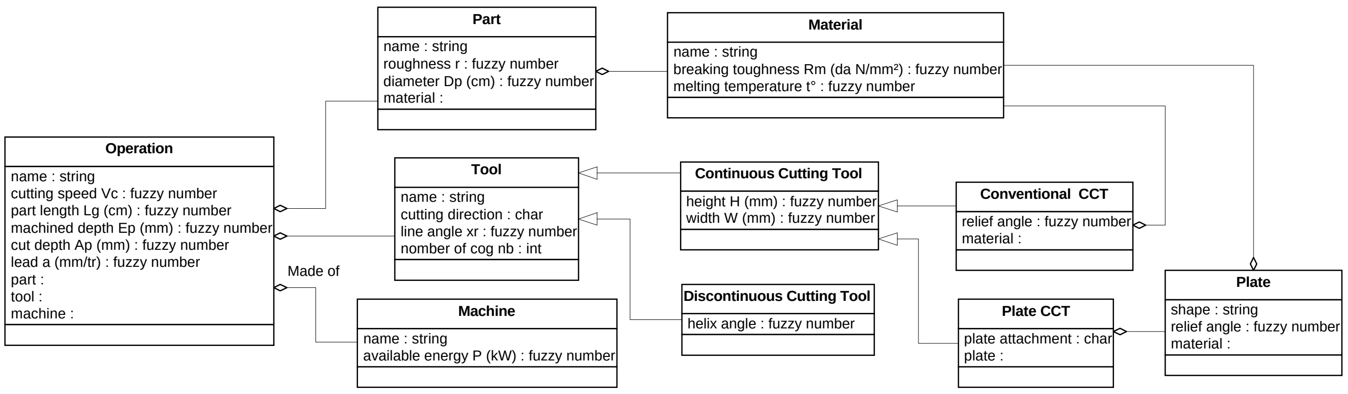
- similarity “close to” defined by

$$\mu\ (x,y) = 1 - \frac{|x-y|}{\Delta} \text{ if } 0 \le |x-y| \le \Delta = 1$$

$$\mu\ (x,y) = 0 \text{ else}$$

- similarity “true/false” defined by

$$\mu\ (x,y) = 1 \text{ if } x = y \qquad \mu\ (x,y) = 0 \text{ else}$$



Part
name : string
roughness r : fuzzy number
diameter Dp (cm) : fuzzy number
material :
Material
name : string
breaking toughness Rm (da N/mm²) : fuzzy number
melting temperature t° : fuzzy number
Operation
name : string
cutting speed Vc : fuzzy number
part length Lg (cm) : fuzzy number
machined depth Ep (mm) : fuzzy number
cut depth Ap (mm) : fuzzy number
lead a (mm/tr) : fuzzy number
part :
tool :
machine :
Tool
name : string
cutting direction : char
line angle xr : fuzzy number
nomber of cog nb : int
Made of
Continuous Cutting Tool
height H (mm) : fuzzy number
width W (mm) : fuzzy number
Conventional CCT
relief angle : fuzzy number
material :
Discontinuous Cutting Tool
helix angle : fuzzy number
Machine
name : string
available energy P (kW) : fuzzy number
Plate CCT
plate attachment : char
plate :
Plate
shape : string
relief angle : fuzzy number
material :

Figure 4. Partial schema of the knowledge base

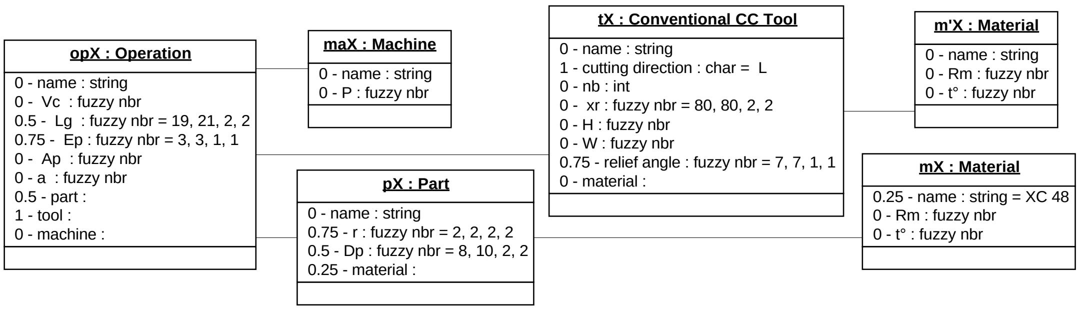


Figure 5. Operation to configure

- "ad hoc" similarity for instance for the comparison of material defined as follows:

μ	XC18	XC25	XC38	XC48	XC60
XC48	0.7	0.8	0.9	1	0.3

- similarity between objects defined as follows for objects o and o':

$$\mu_\beta(o,o') = \beta.\Pi(o,o') + (1-\beta).N(o,o')$$

where β enables to tune the strongness of required similarity. β=1 corresponds to a loose requirement on the similarity whereas β=0 corresponds to a strong requirement on the similarity. In the following, we use this similarity function with β=1.

Results of the similarity and adaptability of each operation of the knowledge base are given in table 1.

Table 1. Similarity and adaptability measures results

	Op1	Op2	Op3	Op4

Similarity with OpX	Π=0.5
	N=0.25
		Π=0.5
		N=0.25
			Π=0.33
			N=0.25
				Π=0.33
				N=0.25

Adaptability	0.734	0.748	0.752	0.744

We can observe that from a similarity point of view, operation Op1 and operation Op2 have the same similarity degrees with operation OpX. Nevertheless operation Op1 can not be easily adapted to configure operation OpX (since its adaptability is equal to 0.734: the worst adaptability), when operation Op2 is more adaptable and therefore is an interesting target for our configuration process. Operation Op3 is more adaptable but is less similar and should therefore not be privileged. The same remark can be done for operation Op4.

4.2 Adaptation

When operation Op2 is chosen, we determine the adaptation domains for the operation at level α = 0.5. The values for the example are given for each attribute on figure 7. Constraints of the domain have now to be propagated on the adaptation domains to produce a solution for the configuration problem. Three binary constraints are taken into account in this example:

- an attachment constraint between a machine and a tool: C_{M-T},
- a compatibility constraint between a tool and its material: C_{T-TM},
- a machining constraint between the name of a tool material and the name of a part material: C_{TM-PM}.

The fuzzy relation R_{TM-PM} defining the constraint C_{TM-PM} is as follows:

$\mu_{R\ TM-PM}(N_{TM},N_{PM}) = 1$ if N_{TM} = N2, N3, N4, N10 and N_{PM} = XC48
$\mu_{R\ TM-PM}(N_{TM},N_{PM}) = 0.9$ if N_{TM} = N1 and N_{PM} = XC48
$\mu_{R\ TM-PM}(N_{TM},N_{PM}) = 0.5$ if N_{TM} = N13 and N_{PM} = XC48
$\mu_{R\ NTM-NPM}(N_{TM},N_{PM}) = 0.4$ if N_{TM} = N11 and N_{PM} = XC48

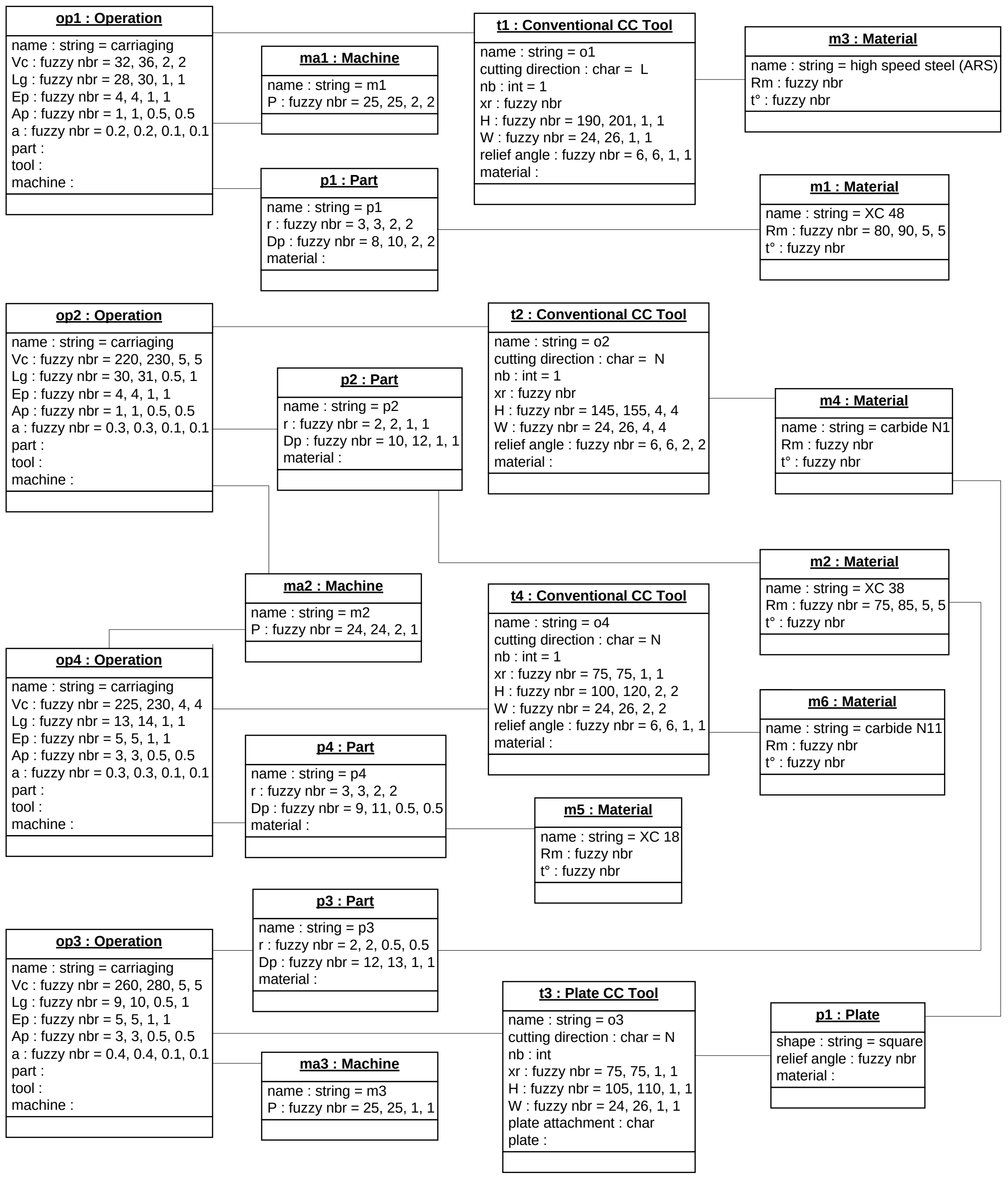
The fuzzy relations defining constraints C_{M-T} and C_{T-TM} are described in table 2.

Table 2. Description of constraints C_{M-T} and C_{T-TM}

C_{M-T}	μ	t_2	t_3	t_4	other
	N1	1	1	0	0
	N11	0	0	1	0
	other	0	0	0	0

C_{T-TM}	μ	t_1	t_2	t_3	t_4
	ma_1	1	0.7	0.6	0
	ma_2	0	1	0.7	1
	ma_3	0	1	1	0

In order to construct the search tree of solutions we choose an order for the instantiation of the variables. The first variable to instantiate is the name of the part material of OpX (PM=XC48). Then we choose to instantiate respectively: the name of the tool material (TM), the tool (T) and the machine (M). At each node of the tree, a value for the variable is chosen and the filtering fuzzy algorithm FAC-3 is applied for reducing variables domains.



op1 : Operation
name : string = carriaging
Vc : fuzzy nbr = 32, 36, 2, 2
Lg : fuzzy nbr = 28, 30, 1, 1
Ep : fuzzy nbr = 4, 4, 1, 1
Ap : fuzzy nbr = 1, 1, 0.5, 0.5
a : fuzzy nbr = 0.2, 0.2, 0.1, 0.1
part :
tool :
machine :
ma1 : Machine
name : string = m1
P : fuzzy nbr = 25, 25, 2, 2
p1 : Part
name : string = p1
r : fuzzy nbr = 3, 3, 2, 2
Dp : fuzzy nbr = 8, 10, 2, 2
material :
t1 : Conventional CC Tool
name : string = o1
cutting direction : char = L
nb : int = 1
xr : fuzzy nbr
H : fuzzy nbr = 190, 201, 1, 1
W : fuzzy nbr = 24, 26, 1, 1
relief angle : fuzzy nbr = 6, 6, 1, 1
material :
m3 : Material
name : string = high speed steel (ARS)
Rm : fuzzy nbr
t° : fuzzy nbr
m1 : Material
name : string = XC 48
Rm : fuzzy nbr = 80, 90, 5, 5
t° : fuzzy nbr
op2 : Operation
name : string = carriaging
Vc : fuzzy nbr = 220, 230, 5, 5
Lg : fuzzy nbr = 30, 31, 0.5, 1
Ep : fuzzy nbr = 4, 4, 1, 1
Ap : fuzzy nbr = 1, 1, 0.5, 0.5
a : fuzzy nbr = 0.3, 0.3, 0.1, 0.1
part :
tool :
machine :
p2 : Part
name : string = p2
r : fuzzy nbr = 2, 2, 1, 1
Dp : fuzzy nbr = 10, 12, 1, 1
material :
t2 : Conventional CC Tool
name : string = o2
cutting direction : char = N
nb : int = 1
xr : fuzzy nbr
H : fuzzy nbr = 145, 155, 4, 4
W : fuzzy nbr = 24, 26, 4, 4
relief angle : fuzzy nbr = 6, 6, 2, 2
material :
m4 : Material
name : string = carbide N1
Rm : fuzzy nbr
t° : fuzzy nbr
m2 : Material
name : string = XC 38
Rm : fuzzy nbr = 75, 85, 5, 5
t° : fuzzy nbr
ma2 : Machine
name : string = m2
P : fuzzy nbr = 24, 24, 2, 1
op4 : Operation
name : string = carriaging
Vc : fuzzy nbr = 225, 230, 4, 4
Lg : fuzzy nbr = 13, 14, 1, 1
Ep : fuzzy nbr = 5, 5, 1, 1
Ap : fuzzy nbr = 3, 3, 0.5, 0.5
a : fuzzy nbr = 0.3, 0.3, 0.1, 0.1
part :
tool :
machine :
t4 : Conventional CC Tool
name : string = o4
cutting direction : char = N
nb : int = 1
xr : fuzzy nbr = 75, 75, 1, 1
H : fuzzy nbr = 100, 120, 2, 2
W : fuzzy nbr = 24, 26, 2, 2
relief angle : fuzzy nbr = 6, 6, 1, 1
material :
m6 : Material
name : string = carbide N11
Rm : fuzzy nbr
t° : fuzzy nbr
p4 : Part
name : string = p4
r : fuzzy nbr = 3, 3, 2, 2
Dp : fuzzy nbr = 9, 11, 0.5, 0.5
material :
m5 : Material
name : string = XC 18
Rm : fuzzy nbr
t° : fuzzy nbr
p3 : Part
name : string = p3
r : fuzzy nbr = 2, 2, 0.5, 0.5
Dp : fuzzy nbr = 12, 13, 1, 1
material :
op3 : Operation
name : string = carriaging
Vc : fuzzy nbr = 260, 280, 5, 5
Lg : fuzzy nbr = 9, 10, 0.5, 1
Ep : fuzzy nbr = 5, 5, 1, 1
Ap : fuzzy nbr = 3, 3, 0.5, 0.5
a : fuzzy nbr = 0.4, 0.4, 0.1, 0.1
part :
tool :
machine :
t3 : Plate CC Tool
name : string = o3
cutting direction : char = N
nb : int
xr : fuzzy nbr = 75, 75, 1, 1
H : fuzzy nbr = 105, 110, 1, 1
W : fuzzy nbr = 24, 26, 1, 1
plate attachment : char
plate :
p1 : Plate
shape : string = square
relief angle : fuzzy nbr
material :
ma3 : Machine
name : string = m3
P : fuzzy nbr = 25, 25, 1, 1

Figure 6. Knowledge base

Before beginning the constraint propagation process, variables domains are as fallow:

$D_M = \{ma_1, ma_2, ma_3\}$ $D_T = \{t_2, t_3, t_4\}$

$D_{TM} = \{N1, N2, N3, N4, N10, N11, N13\}$

$D_{PM} = \{XC18, XC25, XC38, XC48\}$

All the variable values belong initially to their domain with a membership degree to 1.

The first step of the constraint propagation is to choose a value for the name of the part material: PM = XC48 and to apply the FAC-3 algorithm. After that, domain will be modified and also membership degree. FAC-3 algorithm runs to the following domains with membership degrees for the variables:

$D_{PM} = \{XC48\}$, with $\mu_{PM}(XC48) = 0.9$

$D_M = \{ma_1, ma_2, ma_3\}$, with $\mu_M(ma_1) = 0.7$
$\mu_M(ma_2) = 0.9$
$\mu_M(ma_3) = 0.9$

$D_T = \{t_2, t_3, t_4\}$, with $\mu_T(t_2) = 0.9$
$\mu_T(t_3) = 0.9$
$\mu_T(t_4) = 0.4$

$D_{TM} = \{N11, N1\}$, with $\mu_{TM}(N11) = 0.4$
$\mu_{TM}(N1) = 0.9$

The estimated local consistency has a value of 1. The next variable to instantiate is TM. We choose the value TM = N1 with the membership degree of 0.9. After applying the FAC-3 algorithm, we obtain the partial search tree represented in figure 8 (values in brackets correspond to the upper bound of local consistency).

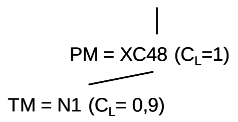


Figure 8. Partial search tree

The complete search tree is shown in figure 9 (intermediate search steps are not developed, the process is the same as previously stated).

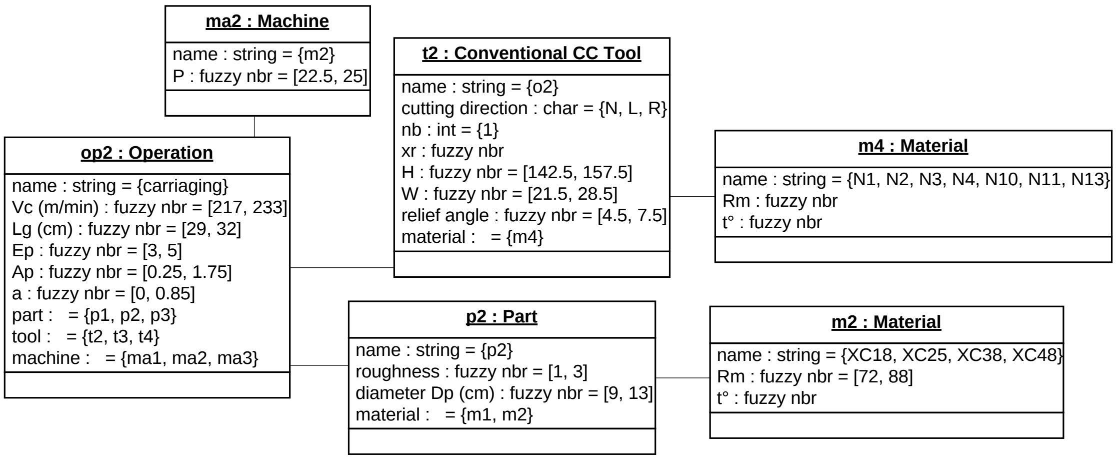

ma2 : Machine
name : string = {m2}
P : fuzzy nbr = [22.5, 25]
t2 : Conventional CC Tool
name : string = {o2}
cutting direction : char = {N, L, R}
nb : int = {1}
xr : fuzzy nbr
H : fuzzy nbr = [142.5, 157.5]
W : fuzzy nbr = [21.5, 28.5]
relief angle : fuzzy nbr = [4.5, 7.5]
material : = {m4}
m4 : Material
name : string = {N1, N2, N3, N4, N10, N11, N13}
Rm : fuzzy nbr
t° : fuzzy nbr
op2 : Operation
name : string = {carriaging}
Vc (m/min) : fuzzy nbr = [217, 233]
Lg (cm) : fuzzy nbr = [29, 32]
Ep : fuzzy nbr = [3, 5]
Ap : fuzzy nbr = [0.25, 1.75]
a : fuzzy nbr = [0, 0.85]
part : = {p1, p2, p3}
tool : = {t2, t3, t4}
machine : = {ma1, ma2, ma3}
p2 : Part
name : string = {p2}
roughness : fuzzy nbr = [1, 3]
diameter Dp (cm) : fuzzy nbr = [9, 13]
material : = {m1, m2}
m2 : Material
name : string = {XC18, XC25, XC38, XC48}
Rm : fuzzy nbr = [72, 88]
t° : fuzzy nbr

Figure 7. Adaptation domains

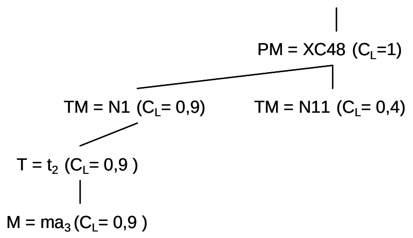


Figure 9. Complete search tree

Thanks to FAC-3 algorithm and CSP propagation techniques, domain constraints are applied on restricted domains and values for some variables are found. We can propose the user a machining operation based on Op2 with some valued characteristics.

5 CONCLUSION

In this paper we describe our work and propositions to base configuration process on past configuration experiences. First of all we propose similarity and adaptability measures for searching among past configuration problems the most similar and adaptable problem to the current problem to solve. Based on this retrieved problem we can reuse its solution and adapt it to the current problem. We define a search space near the retrieved configuration case. Then we suggest to propagate domain constraints on this search space in order to solve the target configuration problem and propose the user an admissible solution.

We illustrate our propositions by an example on the configuration of a machining operation. This kind of configuration is complex and in many cases, experts solve machining configuration problem thanks to their past configuration experiences. So we propose to combine case based reasoning process and CSP techniques for configuration.

REFERENCES

[1] R. Bergmann and W. Wilke, Towards a new formal model of transformational adaptation in case-based reasoning, *Proceedings of the 13th European Conference on Artificial Intelligence, ECAI 98*, Brighton, United Kingdom, ed. H. Prade, pp. 53-57, 1998.

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

[3] P. Cotter and B. Smyth, Personalisation technologies for the Digital TV World, *14th European Conference on Artificial Intelligence, ECAI 2000*, edited by Werner Horn, IOS Press, 2000.

[4] R. Dechter, , and A. Dechter, Structure driven algorithms for truth maintenance, *Artificial Intelligence Journal*, 82, pp. 1-20, 1996.

[5] D. Dubois and H. Prade, *Fuzzy Sets and Systems*. Eds: Academic Press. New York, Fuzzy Logic CDROM Library, 1996.

[6] D. Dubois, H. Fargier and H. Prade, Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty, *Applied Intelligence* (6) pp 287-309, 1996.

[7] H. Fargier, *Problèmes de satisfaction de contraintes flexiles, application à l'ordonnancement de production*, PhD Thesis, IRIT, Toulouse, France, 1994.

[8] E. Gelle, *On the generation of locally consistent solution spaces inmixed dynamic contraint problems*, PhD Thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, 1998.

[9] L. Geneste, M. Ruet and T. Monteiro, Configuration of a machining operation, *14th European Conference on Artificial Intelligence, ECAI 2000, Configuration Workshop*, Berlin, August 2000.

[10] L. Geneste and M. Ruet, Experience based configuration, *17th International Conference on Artificial Intelligence, IJCAI'01, Configuration Workshop*, Seattle, Washington, USA, August 2001.

[11] J. Kolodner, *Case Based Reasoning*, Morgan Kaufmann Publishers, Inc., 1993.

[12] C. Lottaz, *Constraint solving, preference activation and solution adaptation in IDIOM*, Technical report, Artificial Intelligence Laboratory, Swiss Federal Institute of Technology, May/June, 1996.

[13] T. Monteiro, J.L. Perpen, L. Geneste, Configuring a machining operation as a constraint satisfaction problem, *CIMCA'99*, Austria, 17-19 February 1999.

[14] N. Neagu and B. Faltings, Constraint satisfaction for case adaptation, *Workshop on Case Adaptation of the International Conference on Case-based Reasoning, ICCBR'99*, Kaiserslautern, Germany, 1999.

[15] N. Neagu and B. Faltings, *Exploiting Interchangeability Algorithms over Discrete CSPs*, Artificial Intelligence Laboratory (LIA), Computer Science Department, Swiss Federal Institute of Technology (EPFL), 2001.

[16] L. Purvis and P. Pu, An approach to case combination, *Workshop on adaptation in case-based reasoning, ECAI 96*, Budapest, Hungary, 1996.

[17] L. Purvis, Synergy and commonality in case-based and constraint based reasoning, *Proceedings of the AAAI Spring Symposium on Multimodal Reasoning*, Stanford CA, 1998.

[18] D. Sam, *Constraint consistency techniques for continuous domains*, PhD Thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, 1995.

[19] M. H. Sqalli, L. Purvis and E.C. Freuder, Survey of applications integrating constraint satisfaction and case-based reasoning, *PACLP99: The First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming*, London, UK, 1999.

[20] M. H. Sqalli and E.C. Freuder, CBR support for CSP modeling of InterOperability testing, *AAAI-98 Workshop on Case-Based Reasoning Integrations* , Technical Report WS-98-15, pp. 155-160. July 27, 1998, Madison, Wisconsin, USA, 1998.

[21] R. Weigel, B. Faltings and M. Torrens, Interchangeability for Case Adaptation in Configuration Problems. *Workshop on Case-Based Reasoning Integrations (AAAI-98)*, Madison, Wisconsin, USA, 1998.

Distributed generative CSP framework for multi-site product configuration

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

Abstract. Today's configurators are centralized systems and do not allow manufacturers to cooperate on-line for offer-generation or sales-configuration. However, supply chain integration of configurable products requires the cooperation of the configuration systems from the different manufacturers that jointly offer solutions to customers. As a consequence, there is a high potential for methods that enable the computation of such configurations by independent specialized agents. Several approaches to *centralized* configuration tasks are based on constraint satisfaction problem (CSP) solving. Most of them extend the traditional CSP approach in order to comply to the specific expressivity and dynamism requirements for configuration and similar synthesis tasks.
The distributed generative CSP (DisGCSP) framework proposed here builds on a CSP formalism that encompasses the *generative* aspect of variable creation and extensible domains of problem variables. It also builds on the distributed CSP (DisCSP) framework, allowing for approaches to configuration tasks where the knowledge is distributed over a set of agents. Notably, the notions of constraint and nogood are generalized to an additional level of abstraction, extending inferences to types of variables. The usage of the new framework is exemplified by describing modifications to some complete algorithms for DisCSP when targeting DisGCSPs.

1 Introduction/Background

The paradigm of mass-customization allows customers to tailor (configure) a product or service according to their specific needs, i.e. the customer can select between several features and options that should be included in the configured product and can determine the physical component structure of the personalized product variant. Typically, there are several technical and marketing restrictions on the legal parameter constellations and the physical layout. This led manufacturers to develop support for checking the feasibility of user requirements and for computing a consistent solution. This functionality is provided by product configuration systems (configurators), whereby they have shown to be a successful application area for different AI techniques [15] such as description logics [8], or rule-based [1] and constraint-based solving algorithms. [4] describes the industrial use of constraint techniques for the configuration of large and complex systems such as telecommunication switches and [7] is an example of a powerful tool based on Constraint Satisfaction available on the market.
However, companies find themselves in dynamically determined coalitions with other highly specialized solution providers that jointly offer customized solutions. This high integration aspect of todays digital markets implies that software systems supporting the selling and configuration task must no longer be conceived as standalone systems. A product configurator can be therefore seen as an agent with private knowledge that acts on behalf of its company and cooperates with other agents to solve a configuration task. This paper abstracts the *centralized* definition of a configuration task in [16] to a more general definition of a *generative* CSP that is also applicable to the wider range of synthesis problems. Furthermore, we propose a framework that allows to address distributed configuration tasks by extending DisCSPs with the innovative aspects of local generative CSPs:

1. The constraints (and nogoods) are generalized to a form where they can depend on types rather than on identities of variables. This also enables an elegant treatment of the next aspects.
2. The number of variables of certain types that are active in the local CSP of an agent, may vary depending on the state of the search process. In the DisCSP framework, the external variables existing in the system are predetermined, but here the set of variables defining the problem is determined dynamically.
3. The domain of the variables may vary dynamically. Some variables model possible connections and they depend on the existence of components that could become connected.

We also describe the interesting impact of the previously mentioned changes on asynchronous algorithms. In the following we motivate our approach with an example, Section 3 defines a generative CSP and in Section 4 distributed generative CSP is formalized and extensions to current DisCSP frameworks are presented.

2 Motivating example

For the purpose of illustration of our approach we chose as example domain the well known N-queens problem. The characteristics of a distributed configuration problem or similar distributed synthesis tasks are integrated into our N-queens scenario: (a) parts of the problem (i.e., variables) are shared among agents and (b) the problem is dynamically extended (i.e., N is increased), if no solution can be found. Adding additional problem variables leads to domain extensions and thus to a larger search- and solution space. The goal is to place N queens on distinct squares in an $N \times N$ chess board, where no two queens threaten each other [17]. We formalize the problem by making each row of the board a problem variable x_i, where the subscript i ensures unique variable names. In a distributed setting we employ three agents, each owning a fraction of the constraints necessary to solve the N-queens problem. Furthermore, we want to show the *generative* aspect of problem solving in the example, where agents start with a representation of a 0-queens problem and specific requirements on the final solution coming from outside.

[1] Computer Science and Manufacturing, Universität Klagenfurt, Universitätsstrasse 65-67, 9020 Klagenfurt, Austria. e-mail: {felfernig, friedrich, jannach, zanker}@ifit.uni-klu.ac.at

Once the agents determine that a solution cannot be found, they extend the problem space by adding an additional row which in consequence enlarges the domain of row variables by one. Since the exact number of problem variables is not known from the beginning, constraints cannot be directly formulated on concrete variables. Instead, comparable to programming languages, variable types exist that allow to associate a newly created variable with a domain and we can specify relationships in terms of *generic constraints.* [16] define a generic constraint γ as a constraint schema, where meta-variables V^t act as placeholders for concrete variables of a specific type t[2]. In our example three types of problem variables exist, representing the even (t_e) and the uneven rows (t_u) as well as a type (t_c) of counter variables (x_{type}) for the number of instantiations of each type, which allows us to distribute the N-queens constraints among the agents. Therefore, each agent a_i posesses a set of private constraints Γ^{a_i}, i.e., $\Gamma^{a_1} = \{\gamma_1, \gamma_2, \gamma_3, \gamma_8, \gamma_9\}$, $\Gamma^{a_2} = \{\gamma_4, \gamma_5, \gamma_8, \gamma_9\}$ and $\Gamma^{a_3} = \{\gamma_6, \gamma_7, \gamma_8, \gamma_9\}$, that are defined as follows:
$\gamma_1 : val(x_{t_u}) = val(x_{t_e}) \vee val(x_{t_u}) = val(x_{t_e}) + 1$, where $val(x)$ is a predicate that gives the assigned value of variable x.
Informally, the number of uneven rows may exceed the number of even rows by one.
$\gamma_2 : val(V^{t_u}) \neq val(V^{t_e})$
No two queens on an even and an uneven row are allowed to take the same column value.
$\gamma_3 : abs(2 \times (index(V^{t_u}) - index(V^{t_e})) - 1) \neq abs(val(V^{t_u}) - val(V^{t_e}))$, where $index(x)$ returns a number i indicating that x is the i^{th} variable of its type and $abs(n)$ is a predicate that returns the absolute value of n.
No two queens on an even and an uneven row are allowed to be on the same diagonal.
$\gamma_4 : V_1^{t_u} \neq V_2^{t_u} \rightarrow val(V_1^{t_u}) \neq val(V_2^{t_u})$.
No two queens on uneven rows are allowed to take the same column value.
$\gamma_5 : V_1^{t_u} \neq V_2^{t_u} \rightarrow abs(2 \times (index(V_1^{t_u}) - index(V_2^{t_u}))) \neq abs(val(V_1^{t_u}) - val(V_2^{t_u}))$.
No two queens on uneven rows are allowed to be on the same diagonal.
$\gamma_6 : V_1^{t_e} \neq V_2^{t_e} \rightarrow val(V_1^{t_e}) \neq val(V_2^{t_e})$.
No two queens on even rows are allowed to take the same column value.
$\gamma_7 : V_1^{t_e} \neq V_2^{t_e} \rightarrow abs(2 \times (index(V_1^{t_e}) - index(V_2^{t_e})) \neq abs(val(V_1^{t_e}) - val(V_2^{t_e}))$.
No two queens on even rows are allowed to be on the same diagonal.
$\gamma_8 : val(V^{t_u}) \leq x_{t_e} + x_{t_u}$. $\gamma_9 : val(V^{t_e}) \leq x_{t_e} + x_{t_u}$.
The latter two constraints delimit the domain of row variables to the total number of rows.
Figure 1 depicts the initial situation, with a 0-queens problem. The customer requests agent a_1 to satisfy the requirement of finding a solution containing at least two uneven rows:
$\gamma_{cust} : x_{t_u} \geq 2$.
Having added γ_{cust} to the set of private constraints of agent a_1, the search process starts and the solution space is continuously extended by the instantiation of additional problem variables, until a solution is found for a 4-queens problem that satisfies all local constraints of the agents. The links between two agents indicate that they share variables, which is described in more detail later on. Thus, a solution to a generative constraint satisfaction problem requires not only finding valid assignments to variables, but also determining the exact size of the problem itself. In the sequel of the paper we define a model for the local configurators and we detail extensions to DisCSP algorithms.

[2] The exact semantics of generic constraints is given in Definition 2 in Section 3.

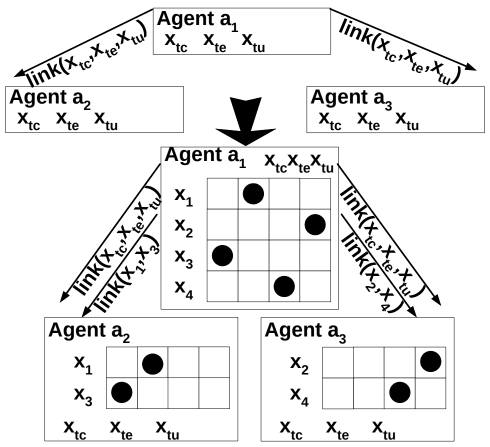


Figure 1. Motivating example

3 Generative Constraint Satisfaction

In many applications, solving is a *generative* process, where the number of involved components (i.e., variables) is not known from the beginning. To represent these problems we employ an extended formalism that complies to the specifics of configuration and other synthesis tasks where problem variables representing components of the final system are *generated* dynamically as part of the solution process because their total number cannot be determined beforehand. The framework is called *generative* CSP (GCSP) [5, 16]. This kind of dynamicity extends the approach of dynamic CSP (DCSP) formalized by Mittal and Falkenhainer [9], where all possibly involved variables are known from the beginning. This is needed because the activation constraints reason on the variable's activity state. [10] propose a conditional CSP to model a configuration task, where structural dependencies in the configuration model are exploited to trigger the activation of subproblems. Another class of DCSP was first introduced by [3] where constraints can be added or removed independently of the initial problem statement. The dynamicity occuring in a GCSP differentiates from the one described in [3] in the sense that a GCSP is extended in order to find a consistent solution and the latter has already a solution and is extended due to influence from the outside world (e.g., additional constraints) that necessitates finding a new solution. Here we give a definition of a GCSP that abstracts from the configuration task specific formulation in [16] and applies to the wider range of synthesis problems.

Definition 1 (Generative constraint satisfaction problem (GCSP)) *A generative constraint satisfaction problem is a tuple GCSP(X, Γ, T, Δ), where:*

- *X is the set of problem variables of the GCSP and $X_0 \subseteq X$ is the set of initially given variables.*
- *Γ is the set of generic constraints.*
- *$T = \{t_1, \ldots, t_n\}$ is the set of variable types t_i, where $dom(t_i)$ associates the same domain to each variable of type t_i, where the domain is a set of atomic values.*

- *For every type $t_i \in T$ exists a counter variable $x_{t_i} \in X_0$ that holds the number of variable instantiations for type t_i. Thus, explicit constraints involving the total number of variables of specific types and reasoning on the size of the CSP becomes possible.*
- *Δ is a total relation on $X \times (T, N)$, where N is the set of positive integer numbers. Each tuple $(x, (t, i))$ associates a variable $x \in X$ with a unique type $t \in T$ and an index i, that indicates x is the i^{th} variable of type t. The function $type(x)$ accesses Δ and returns the type $t \in T$ for x and the function $index(x)$ returns the index of x.*

By generating additional variables, a previously unsolvable CSP can become solvable, which is explained by the existence of variables that hold the number of variables.

When modeling a configuration problem, variables representing named connection points between components, i.e., *ports*, will have references to other ports as their domain. Consequently, we need variables whose domain varies depending on the size of a set of specific variables [16].

Example Given t_{mod} as the type of variables representing ports of *modules* and t_{port} as the type of *port* variables that are allowed to connect to *modules*, then the domain of the *port* variables $dom(t_{port})$ must contain references to *modules*. This is specified by defining $dom(t_{port}) = \{1, \ldots, ub\}$, where ub is an upperbound on the number of variables of type t_{mod}, and formulating an additional generic constraint that restricts all variables of type t_{port} using the counter variable for the total number of variables having type t_{mod}, i.e., $val(V^{t_{port}}) \leq x_{t_{mod}}$. With the help of the $index()$ function concrete variables can then be referenced.

Referring to our introductory example we can formalize the local GCSP of agent a_1 (initially consisting only of counter variables x_{t_i}, their type t_c, and the types of row variables) as $X^{a_1} = \{x_{t_c}, x_{t_e}, x_{t_u}\}$, $\Gamma^{a_1} = \{\gamma_1, \gamma_2, \gamma_3, \gamma_8, \gamma_9\}$, $T^{a_1} = \{t_c, t_e, t_u\}$ and $\Delta^{a_1} = \{(x_{t_c},(t_c, 1)),(x_{t_e},(t_c, 2)), (x_{t_u},(t_c, 3))\}$. The domain for even and uneven row variables is consequently defined as $dom(t_e) = dom(t_u) = dom(t_c) = \{1, \ldots, ub\}$, where the domains for the row variables are limited by the domain constraints (i.e., γ_8, γ_9).

Definition 2 (Generic constraint) *A generic constraint $\gamma \in \Gamma$ formulates a restriction on the meta-variables $M_a, \ldots, M_k$. A meta-variable M_i is associated a variable type $type(M_i) \in T$ and must be interpreted as a placeholder for all concrete variables x_j, where $type(x_j) = type(M_i)$.*

Note, that generic constraints can also formulate restrictions on specific initial variables from X_0 by employing the $index()$ function.

Consider the GCSP(X, Γ, T, Δ) and let $\gamma \in \Gamma$ restrict the meta-variables $M_a, \ldots, M_k$, where $type(M_i) \in T$ is the defined variable type of the meta variable M_i.

Definition 3 (Consistency of generic constraints) *Given an assignment tuple θ for the variables X, then γ is said to be satisfied under θ, iff*
$\forall x_a, \ldots, x_k \in X : type(x_a) = type(M_a) \wedge \ldots \wedge type(x_k) = type(M_k) \rightarrow \gamma[M_a|_{x_a}, \ldots, M_k|_{x_k}]$ is satisfied unter θ, where $M_i|_{x_i}$ indicates that the meta-variable M_i is substituted by the concrete variable x_i.

Thus a *generic* constraint must be seen as a constraint scheme that is expanded into a set of constraints after a preprocessing step, where meta-variables are replaced by all possible combinations of concrete variables having the same type, e.g., given a GCSP of agent a_1 (excluding counter variables) with $X^{a_1} = \{x_1, x_2, x_3\}$, $T^{a_1} = \{t_u, t_e\}$ and $\Delta^{a_1} = \{(x_1, (t_u, 1)), (x_2, (t_e, 1)), (x_3, (t_u, 2))\}$, the satisfiability of the generic constraint γ_2 is checked by testing the following conditions: $val(x_1) \neq val(x_2)$. $val(x_3) \neq val(x_2)$.

Definition 4 (Solution for a generative CSP) *Given a generative constraint satisfaction problem GCSP(X_0, Γ, T, Δ_0), then its solution encompasses the finding of a set of variables X, type and index assignments Δ and an assignment tuple θ for the variables in X, s.t.*

1. *for every variable $x \in X$ an assignment $x = v$ is contained in θ, s.t. $v \in dom(type(x))$ and*
2. *every constraint $\gamma \in \Gamma$ is satisfied under θ and*
3. *$X_0 \subseteq X \wedge \Delta_0 \subseteq \Delta$.*

Note, that we do not impose a minimality criterium on the number of variables in our solution, because in practical applications different optimization criteria exist, such as total cost or flexibility of the solution, thus non-minimal solutions can be preferred over minimal ones.

The calculated solution (excluding counter variables) for the local GCSP of agent a_1 consists of $X^{a_1} = \{x_1, x_2, x_3, x_4\}$, $\Delta^{a_1} = \{(x_1, (t_u, 1)),(x_2, (t_e, 1)),(x_3, (t_u, 2)),(x_4, (t_e, 2))\}$ and the assignment tuple $x_1 = 2$, $x_2 = 4$, $x_3 = 1$ and $x_4 = 3$. Thus, $x_1, \ldots, x_4$ are the names of *generated* variables.

Note, that names for generated variables are unique and can be randomly chosen by the GCSP solver implementation and therefore constraints must not formulate restrictions on the variable names of generated variables. Consequently, substitution of any generated variable (i.e., $x \in X \setminus X_0$) by a newly generated variable with equal type, index and value assignment has no effect on the consistency of generic constraints. Our GCSP definition extends the definition from [16] in the sense that a finite set of variable types T is given and during problem solving variables having any of these types can be generated, whereas in [16] only variables of a single type, i.e., component variables, can be created. Current CSP implementations of configuration systems (e.g., [7] [4]) use a type system for problem variables, where new variable instances, having one of the predefined types, are dynamically created. This is only indirectly reflected in the definition of [16] by the domain definition of component variables, which we explicity represent as a set of types. Furthermore, the definition of *generic constraints* does not enforce the use of a specific constraint language for the formulation of restrictions. Examples are the LCON language used in the COCOS project [16], or the configuration language of the ILOG Configurator [7].

Note, that the set of variables X can be theoretically infinite, leading to an infinite solution space. For practical reasons, solver implementations for a GCSP put a limit on the total number of problem variables to ensure decidability and finiteness of the search space. This way a GCSP is reduced to a dynamic CSP and in further consequence to a CSP. A DCSP models each search state as a static CSP, where complex activation constraints are required to ensure the alternate activation of variables depending on the search state. These constraints need to be formulated for every possible state of the GCSP, which leads to combinatorial explosion of concrete constraints. Furthermore, the formulation of large configuration problems as a DCSP is merely impractical from the perspective of knowledge representation, which is crucial for knowledge-based applications such as configuration systems.

4 DisCSP Framework

In our framework, we are interested only in algorithms that guarantee a good/optimal solution. The first asynchronous complete search algorithm is Asynchronous Backtracking (ABT) [18]. [2] shows how

ABT can be adapted to networks where not all agents can directly communicate to one another. [6] makes the observation that versions of ABT with polynomial space complexity can be designed. The extension of ABT with asynchronous maintenance of consistencies, and asynchronous dynamic reordering is described in [12, 14]. [11] achieves an increased level of abstraction in DisCSPs by letting nogoods (i.e. certain constraints) consist of aggregates (i.e. sets of variable assignments), instead of simple assignments.
We show how the basic DisCSP framework for ABT with extensions for private constraints [11] can be applied to a scenario of distributed product configuration. Therefore, improving the performance of ABT with extensions as referenced above is straightforward. We summarize in the following the properties of the ABT algorithm that guarantee its correctness and completeness [18]. Then we apply this DisCSP framework to a scenario where each agent locally solves a generative constraint satisfaction task. Each time an agent extends the solution space of his local GCSP by creating an additional variable, the DisCSP setting is transformed into a new DisCSP setting, which again has all properties required by asynchronous search to correctly function.

4.1 Asynchronous Search

We summarize the characteristics of asynchronous search algorithms like ABT [18] and its extensions [11] for private constraints:

1. $A = \{a_1, \dots a_n\}$ is a set of n totally ordered agents, where a_i has priority over a_j if $i < j$.
2. Each agent a owns a set of local constraints Γ^a and a is *interested in* those variables that are contained in its local constraints, called *local variables.* A *link* exists between two agents if they share a variable, that is directed from the agent with higher priority to the agent with lower priority. A link from agent a_1 to agent a_2 is refered to as an *outgoing link* of a_1 and an *incoming link* of a_2.
3. An aggregate is a triplet (x_j, set_j, h_j), where x_j is a variable, set_j a set of values for x_j and h_j is a *history* of the pair (x_j, set_j), where the history marks the aggregate with the information required for a correct message ordering (a counter in ABT).
4. The *view* of an agent a is a set of the aggregates for those variables agent a is interested in.
5. The agents communicate using the following types of messages, where channels without communication loss are assumed:
   - **ok?** message. Agents with higher priorities communicate via **ok?** messages a proposal for a set of variables to lower priority agents. Each proposal is associated with a history, that allows the recipient to identify the most recent message.
   - **nogood** message. In case an agent cannot find a proposal that does not violate its own constraints and its stored nogoods, it generates an explanation under the form of an explicit nogood $\neg N$. A nogood can be interpreted as a constraint that forbids a combination of value assignments to a set of variables. It is announced via a **nogood** message to the lowest priority agent that has proposed an assignment[3] in N.
   - **addlink** message. It transports a set of variables $vars$, where the receiver agent is informed that the sender is interested in the variables $vars$ and for every variable in $vars$ a *link* is established from the higher priority agent to the agent with lower priority.

[3] aggregate in the Asynchronous Aggregation Search (AAS) algorithm [11]

6. A *system agent* is a special agent that receives the subscriptions of the agents for the search. Its task is to decide the order of the agents, initialize the links and announce the termination of the search.

4.2 Framework for DisGCSP

A distributed configuration problem is a multi-agent scenario, where each agent wants to satisfy a local GCSP and agents keep their constraints private for security and privacy reasons, but share all variables which they are interested in. As constraints employ meta-variables, the *interest* of an agent in variables needs to be redefined:

Definition 5 (Interest in variables) *An agent a_j owning a local $GCSP^{a_j}$(X^{a_j},Γ^{a_j},T^{a_j},Δ^{a_j}) is said to be interested in a variable $x \in X^{a_h}$ of an agent a_h, if there exists a generic constraint $\gamma \in \Gamma^{a_j}$ formulating a restriction on the meta-variables $M_a, \dots, M_k$, where $type(M_i) \in T^{a_j}$ is the defined variable type of the meta variable M_i, and $\exists M_i \in M_a, \dots, M_k : type(x) = type(M_i)$.*

Definition 6 (Distributed generative CSP) *A distributed generative constraint satisfaction problem has the following characteristics:*

- *$A = \{a_1, \dots, a_n\}$ is a set of n agents, whereby each agent a_i owns a local $GCSP^{a_i}$(X^{a_i},Γ^{a_i},T^{a_i},Δ^{a_i}).*
- *All variables in $\bigcup_{i=1}^{n} X^{a_i}$ and all type denominators in $\bigcup_{i=1}^{n} T^{a_i}$ share a common namespace, ensuring that a symbol denotes the same variable, resp. the same type, with every agent.*
- *For every pair of agents $a_i, a_j \in A$ and for every variable $x \in X^{a_j}$, where agent a_i is interested in x, must hold $x \in X^{a_i}$.*
- *For every pair of agents $a_i, a_j \in A$ and for every shared variable $x \in X^{a_i} \cap X^{a_j}$ the same type and index must be associated to x in the local GCSPs of the agents, i.e., $type^{a_i}(x) = type^{a_j}(x) \wedge index^{a_i}(x) = index^{a_j}(x)$.*

For every pair of agents $a_i, a_j \in A$ and for every shared variable $x \in X^{a_i} \cap X^{a_j}$ a *link* must exist that indicates that they share variable x. The *link* must be directed from the agent with higher priority to the agent with lower priority.

Definition 7 *Given a distributed generative constraint satisfaction problem among a set of n agents then its solution encompasses the finding of a set of variables $X = \bigcup_{i=1}^{n} X^{a_i}$, type and index assignments $\Delta = \bigcup_{i=1}^{n} \Delta^{a_i}$ and an assignment tuple $\theta = \bigcup_{i=1}^{n} \theta^{a_i}$ for every variable in X, s.t. for all agents $a_i : X^{a_i}$, Δ^{a_i} and θ^{a_i} are a solution for the local $GCSP^{a_i}$ of agent a_i.*

Remark A solution to a distributed generative CSP is also a solution to a centralized GCSP($\bigcup_{i=1}^{n} X^{a_i}$, $\bigcup_{i=1}^{n} \Gamma^{a_i}$, $\bigcup_{i=1}^{n} T^{a_i}$, $\bigcup_{i=1}^{n} \Delta^{a_i}$).

Definition 8 (Generic aggregate) *A generic aggregate is a unary generic constraint. It takes the form: $\langle M, i, s, h\rangle$, where M is a meta-variable, i is a set of index values for which the constraint applies, s is a set of values and h is a history of the aggregate.*

Definition 9 (Generic nogood) *A generic nogood takes the form $\neg N$, where N is a set of generic aggregates for distinct meta-variables.*

Given the characteristics of a DisGCSP (see Definition 6) the links can be initialized before the start of the algorithm, due to the common naming space for type denominators and the condition of a unique type and index assignment to variables over all agents.

Value assignments to variables are communicated to agents via **ok?** messages that transport *generic aggregates* in our DisGCSP framework, which represent domain restrictions on variables by unary constraints. Each of these unary constraints in our DisGCSP has attached an unique identifier called constraint reference (cr) [13]. Any inference has to attach the crs associated to arguments into the obtained nogood. We treat the extension of the domains of the variables as a constraint relaxation [13]. For this reason we introduce the next features for algorithm extensions:

- **announce** message broadcasts a tuple (x, t, i), where x is a newly created variable of type t and with index i to all other agents. The receiving agents determine their interest in variable x and react depending on their interest and priority in one of the following ways (a) send an **addlink** message transporting the variable set $\{x\}$ (b) add the sending agent to its outgoing links or (c) discard the message.
- **domain** message broadcasts a set CR of obsolete constraint references. Any receiving agent removes all the nogoods having attached to them a constraint reference $cr \in CR$. The receiver of the message calls then the function *check_agent_view()* detailed in [18], making sure that it has a consistent proposal or that it generates nogoods.
- **nogood** messages transport *generic nogoods* $\neg N$ that contain assignments for meta-variable instances. These messages are multicasted to all agents interested in $\neg N$. An agent A_i is interested in a generic nogood $\neg N$ if it has *interest in* any meta-variable in $\neg N$.
- When an agent needs to revoke the creation of a new variable due to backtracking in his local solving algorithm, he assigns it a specific value from its domain indicating the deactivation of the variable and communicates it via an **ok?** message to all interested agents.

In order to avoid too many messages a broker agent can be introduced that maintains a static list of agents and their interest in variables of specific types comparable to a *yellow pages* service. In this case the agent that created a new variables only needs to request the broker agent for a list of interested agents and does not need to broadcast an **announce** message to all agents.

Theorem 1 *Whenever an existing extension of ABT is extended with the previous messages and is applied to DisGCSPs, the obtained protocols are correct, complete and terminate.*

Proof: Let us consider that we extend a protocol called P.
Completeness: All the generated information results by inference. If failure is inferred (when no new component is available), then indeed no solution exists.
Termination: Without introducing new variables, the algorithm terminates. Since the number of variables that can be generated is finite, termination is ensured.
Correctness: The resulting overall protocol is an instance of P, where the delays of the system agent initializing the search equals the time needed to insert all the variables generated before termination. Therefore the result satisfies all the agents and the solution is correct □

5 Conclusions

Building on the definition of a centralized configuration task from [16], we formally defined a new class of CSP, termed generative CSP (GCSP), that generalizes the approaches of constraint-based configurator applications in use [4, 7]. The innovative aspects include an additional level of abstraction for constraints and nogoods. Constraints and nogoods can refer to types of variables. Furthermore, we extended GCSP to a distributed scenario, where this abstraction adapts well DisCSP frameworks for dynamic configuration problems (but it can be used in static models as well). We have described how this enhancement can be naturally integrated in a large family of existing asynchronous algorithms for DisCSPs.

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] C. Bessière, A. Maestre, and P. Meseguer, 'Distributed dynamic backtracking', in *Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, p. 772, Paphos, Cyprus, (2001).

[3] R. Dechter and A. Dechter, 'Belief Maintenance in Dynamic Constraint Networks', in *Proc. 7th National Conf. on Artificial Intelligence (AAAI)*, pp. 37–42, St. Paul, MN, (1988).

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

[5] A. Haselböck, *Knowledge-based configuration and advanced constraint technologies*, Ph.D. dissertation, Technische Universität Wien, 1993.

[6] W. Havens, 'Nogood caching for multiagent backtrack search', in *Proc. of 14th National Conf. on Artificial Intelligence (AAAI), Agents Workshop*, Providence, Rhode Island, (1997).

[7] D. Mailharro, 'A classification and constraint-based framework for configuration', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **12(4)**, 383–397, (1998).

[8] D.L. McGuiness and J.R. Wright, 'Conceptual Modeling for Configuration: A Description Logic-based Approach.', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **12(4)**, 333–344, (1998).

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

[10] D. Sabin and E.C. Freuder, 'Configuration as Composite Constraint Satisfaction', in *Proc. of AAAI Fall Symposium on Configuration*, Cambridge, MA, (1996). AAAI Press.

[11] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings, 'Asynchronous search with aggregations', in *Proc. of 17th National Conf. on Artificial Intelligence (AAAI)*, pp. 917–922, Austin, TX, (2000).

[12] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings, 'ABT with asynchronous reordering', in *Proc. of Intelligent Agent Technology (IAT)*, pp. 54–63, Maebashi, Japan, (October 2001).

[13] M.-C. Silaghi, D. Sam-Haroud, and B.V. Faltings, 'Maintaining hierarchically distributed consistency', in *Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP), DCS Workshop*, pp. 15–24, Singapore, (2000).

[14] M.-C. Silaghi, D. Sam-Haroud, and B.V. Faltings, 'Consistency maintenance for ABT', in *Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pp. 271–285, Paphos, Cyprus, (2001).

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

[16] M. Stumptner, G. Friedrich, and A. Haselböck, 'Generative constraint-based configuration.', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **12(4)**, 307–320, (1998).

[17] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[18] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, 'Distributed constraint satisfaction for formalizing distributed problem solving', in *Proc. of 12th Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 614–621, Yokohama, Japan, (1992).

Semantic Configuration Web Services in the CAWICOMS Project

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

Abstract. Product configuration is a key technology in today's highly specialized economy. Within the scope of state-of-the-art B2B frameworks and eProcurement solutions, various initiatives take into account the provision of configuration services. However, they all are based on the idea of defining quasi-standards[2] for many-to-many relationships between customers and vendors. When moving towards networked markets, where suppliers dynamically form supply-side consortia, more flexible approaches to B2B integration become necessary. The emerging paradigm of Web services has therefore a huge potential in business application integration. This paper presents an application scenario for configuration Web services, that is currently under development in the research project CAWICOMS[3]. An ontology-based approach allows the advertisement of services and a configuration specific protocol defines the operational processes. However, the lack of widely adopted standards for the semantic annotation of Web services is still a major shortcoming of current Web technology.

1 Introduction

The easy access to vast information resources offered by the World Wide Web (WWW) opens new perspectives for conducting business. State-of-the-art electronic marketplaces enable many-to-many relationships between customers and suppliers, thus replacing inflexible one-to-one relations dating to the pre-internet era of EDI (electronic data interchange). The problem of heterogeneity of product and catalogue descriptions as well as inter-company process definitions is potentially resolved by imposing a common standard on all market participants, although this can be costly to achieve. In this regard, the non-existence of a single standard for conducting B2B electronic commerce constitutes a major obstacle towards innovation. Examples for competing and partly incompatible B2B frameworks are OBI, RosettaNet, cXML or BizTalk [25]. They all employ XML[4] as a flexible data format definition language, that allows to communicate tree structures with a linear syntax; however, content transformation between those catalog and document standards is far from being a trivial task [8]. The issue of marketplace integration mechanisms for *customizable* products is far more complex, because products have characterizing attributes that offer a range of different choices. Customers are enabled to configure goods and services according to their individual needs at no extra cost following the paradigm of *mass customization* [23]. Product configuration systems (configurators) support sales engineers and customers in coping with the large number of possible product variants.

The goal of the research project CAWICOMS is to enable configuration systems to deal simultaneously with configurators of multiple suppliers over the Web. This allows for end-to-end selection, ordering and provisioning of complex products and services supplied by an extended value chain. We employ an ontology-based approach that builds on the flexible integration of these configuration Web services. Furthermore, it can be shown how the capability of each configuration system can be described on the semantic level using an application scenario from the telecommunication domain. For representation of the semantic descriptions the evolving language standard of the 'Semantic Web' initiative [3], [12], OIL resp. DAML+OIL [9] is employed.

In Section 2 we start by giving an overview on the application domain. In Section 3 we describe the Web service architecture and in Section 4 a multi-layer ontology definition for our application domain is given. The interaction processes between the Web service providers and requestors are discussed in Section 5.

2 Application scenario

Easy access to the corporate network and secure connections to business partners is crucial in today's economy. Virtual Private Networks (VPN) extend the intranet of a possibly multi-national company and are capable of meeting the access requirements at reduced cost using the worldwide IP network services and dedicated service provider IP backbones. VPN infrastructures are designed to be flexible and configurable in order to be able to cope with a rich variety of possible customer requirements. Therefore, the establishment of some concrete VPN involves different steps after determination of customer requirements like locations to be connected or specification of required bandwidth: selection of adequate access facilities from the customer site to some entry point to the VPN backbone, reservation of bandwidth within the backbone, as well as configuration of routing hardware and additional services like installation support.

Note, that it is very unlikely that all these products and services needed for the provision of such a VPN can be supplied by one single organization, but are in general made available by different specialized solution providers, e.g., Internet Service Providers, telecommunication companies or hardware manufacturers (see Figure 1). Therefore, VPNs are typically marketed by specialized resellers (or telecommunication companies like two of our application partners)

[1] Computer Science and Manufacturing, Universität Klagenfurt, Universitätsstrasse 65-67, 9020 Klagenfurt, Austria. e-mail: {felfernig, friedrich, jannach, zanker}@ifi t.uni-klu.ac.at

[2] Due to a high number of competing standardization efforts, none of them appears to be a de-facto industry standard.

[3] CAWICOMS is the acronym for "Customer-Adaptive Web Interface for the Confi guration of Products and Services with Multiple Suppliers". This work was partly funded by the EC through the IST Programme under contract IST-1999-10688 (http://www.cawicoms.org).

[4] See http://www.w3c.org/xml for reference.

that integrate the services of individual suppliers and offer complete VPN solutions to their customers.

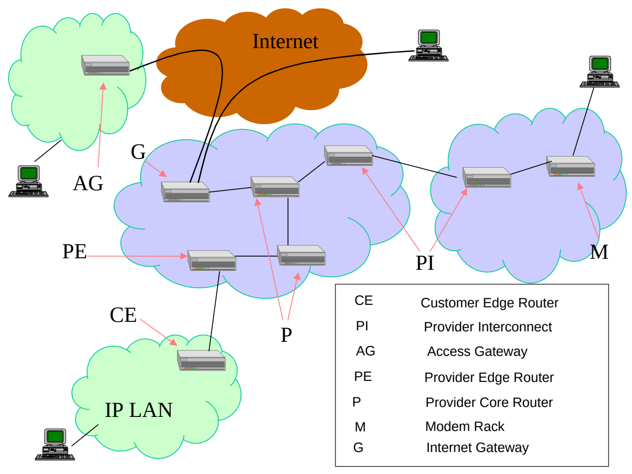


Figure 1. IP-VPN sketch

The integrator/reseller company contracts with the customer and determines - according to the geographic location of the different sites and the qualitative requirements with regards to bandwidth, quality of service or cost limits - the layout of the network service. This configuration task includes the selection of adequate access facilities from the customer site to some entry point of a VPN backbone, reservation of bandwidth within the backbone, as well as parameter setting for routing hardware and configuration of additional services like installation support. Considerable parts of this service package will then be sourced from the specialized solution providers [7].

3 CAWICOMS environment

In the given application scenario, problem solving capabilities are distributed over several business entities that need to cooperate on a customer request for joint service provision. This Peer-to-Peer (P2P) interaction approach among a dynamic set of participants without a clear assignment of *client* and *server* roles asks for applying the paradigm of *Web services* [17]. It stands for encapsulated application logic that is open to accept requests from any peer over the Web.

3.1 Web services

Basically, a Web Service can be defined as an interface that describes a collection of provided operations. In the following we interpret the application logic that configures a product as a standardized Web service. It can be utilized by interface agents interacting with human users in a Web shop as well as by agents that outsource configuration services as part of their problem solving capabilities. When implementing a Web Service the following issues need to be addressed [17]:

- *Service publishing* - the provider of a service publishes the description of the service to a service registry which in our case are configuration agents with mediating capabilities. Within this registry the basic properties of the offered configuration service have to be defined in such a way that automated identification of this service is possible.
- *Service identification* - the requestor of a service imposes a set of requirements which serve as the basis for identifying a suitable service. In our case, we have to identify those suppliers, that are capable of supplying goods or services that match the specific customer requirements.
- *Service execution* - once a suitable service has been identified the requirements need to be communicated to the service agent that can be correctly interpreted and executed. UDDI, WSDL, and SOAP are the evolving technological standards that allow the invocation of remote application logic based on XML syntax.

Following the vision behind the Semantic Web effort [3, 12], the sharing of semantics is crucial to enable the WWW for applications. In order to have agents automatically searching, selecting and executing remote services, representation standards are needed that allow the annotation of meaning of a Web service which can then be interpreted by agents with the help of ontologies.

3.2 Ontologies

In order to define a common language for representing capabilities of configurable products and services we use a hierarchical approach of related ontologies [11, 4]. Ontologies are employed to set a semantic framework that enables the semantic description of Web services in the domain of product configuration. Furthermore, we follow the proposal of [10] to structure the ontological commitments into three hierarchy levels (see Figure 2), namely the *generic ontology level*, the *intermediate level* and the *domain level*.

- *Generic ontology level* - Most modeling languages include some kind of meta-model for representing classes and their relationships (e.g. the frame ontology of Ontolingua [11], the UML meta-model [24] or the representation elements of ontology languages such as OIL or DAML+OIL). Such a meta-model can be interpreted as a generic level ontology. Example modeling concepts included in those ontologies are frame, class, relation, association, generalization, etc.
- *Intermediate ontology level* - the basic modeling concepts formulated on the generic ontology level can be refined and used in order to construct an intermediate ontology which includes widespread modeling concepts used in the domain. Such an ontology for the configuration domain is discussed in [26] who introduce component types, function types, port types, resources and different kinds of constraints as basic configuration domain specific modeling concepts.
- *Domain ontology level* - finally, using the modeling concepts of the intermediate level, we are able to construct application domain specific ontologies (e.g. network services), which can also be denoted as a configuration models.

Note, that similar approaches to structure ontologies are already implemented in a set of ontology construction environments (e.g. [11]). Our contribution in this context is to illustrate their application for integrating configuration systems.

3.3 Interaction scenario

In the following we sketch our Web service scenario that focuses on enabling automated procurement processes for customisable items (see Figure 2). Basically there exist two different types of agents, those that only offer configuration services (L) and those that act as suppliers as well as requestors for these services (I). The denotation of agent types derives from viewing the informational supply chain of product configuration as a tree[5], where a configuration system constitutes either an *inner node* (I) or a *leaf node* (L). Agents of type I have therefore the mediating functionality incorporated, that allows the offering agents to advertise their configuration services. Matchmaking for service identification is performed by the mediating capability that is internal to each configurator at an inner node. It is done on the semantic level that is eased by multi-layered ontological commitments (as discussed in the preceding subsection) among participants. It is assumed that suppliers share application domain ontologies that allow them to describe the capabilities of their offered products and services on the semantic level. An approach that abstracts from syntactical specifics and proposes a reasoning on the semantic level also exists for transforming standardized catalog representations in [8]. Abstract service descriptions can be interpreted as physical sub-structures of the product or as a kind of standardized functional description of the product[6]. Furthermore, agents in the role of customers (service requestors) can impose requirements on a desired product; these requirements can be matched against the functional product description provided by the suppliers (service providers). If one or more supplier descriptions match with the imposed requirements, the corresponding configuration service providers can be contacted in order to finally check the feasibility of the requirements and generate a customized product/service solution.

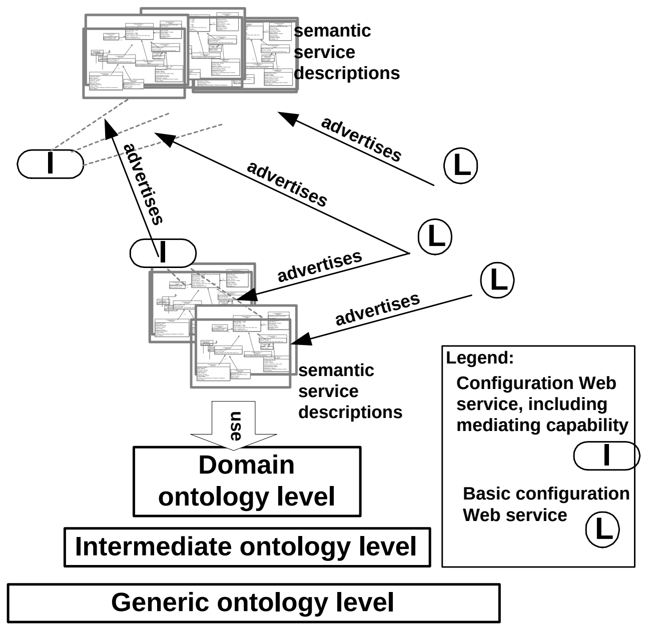

semantic service descriptions
advertises
advertises
advertises
advertises
advertises
I
I
L
L
L
semantic service descriptions
use
Domain ontology level
Intermediate ontology level
Generic ontology level
Legend:
Configuration Web service, including mediating capability
I
Basic configuration Web service
L

Figure 2. Web service scenario

[5] Note, that only the required configuration services are organized in a tree structure, which must not hold for the involved companies in the value chain of a product.

[6] In [18] this kind of description is denoted as a functional architecture of the configured product.

4 Multi-layer Ontology Definition

As sketched in Figure 2 the semantic descriptions of the offered configuration services are based on the three layer approach of [10]. The creation of service profiles for each involved configuration system is supported by a set of knowledge acquisition tools, that allow the definition of the product structure with a graphical UML-based notation with precise semantics [5]. Using translators these implementation independent models are translated into proprietary knowledge bases of problem solving engines such as the Java-based *JConfigurator* from ILOG[7] [14].

However, in the following we will describe our approach employing DAML+OIL as a language for the Semantic Web with precise model theoretic semantics. The correspondence between representation concepts needed for modeling configuration knowledge bases and DAML+OIL is shown in [6]. The uppermost layer of our ontology is the *generic ontology level*. At this level the basic representation concepts and ontological modeling primitives are introduced. These are inherent to the concepts of the modeling language such as *class* and *slot* definitions in OIL. Therefore, it meets the expectations towards the uppermost layer and in the following subsection we move on to show which configuration domain specific modeling primitives are to be provided on the *intermediate ontology level*. For reasons of readability *OIL text* [1] is used for representation, i.e. no RDFS-based representation of DAML+OIL is used.

4.1 Basic Configuration Ontology

A general ontology for the configuration domain is important in order to allow easy configuration knowledge reuse and the integration of complex configurable products within marketplace environments. The ontologies proposed by [26] and [5] serve as a basis for the construction of application domain specific ontologies which allow the description of configuration services on a semantic level. Refined concepts of *classes* such as *component types*, *resource types*, *port types*, or *function types* are the basic modeling concepts useful for building the basic product structure. The ontology defined in [26] is based on the frame ontology of Ontolingua [11] and represents a synthesis of resource-based, function- based, connection-based, and structure-based approaches for representing configuration problems. A similar set of concepts is discussed in [5], where the configuration ontology is represented as a UML profile with additional first order formalizations guaranteeing a precise semantics for the provided modeling concepts.

4.2 Product Domain Ontology for Network Services

While the intermediate configuration ontology contains only the basic concepts for modeling product structures, it allows the construction of more specialized ontologies for specific application domains. Furthermore, axioms and slot constraints provided in OIL can be employed to formulate constraints on the configuration model. Exactly these concepts will be refined in the following for representing (application) domain specific ontologies that can be interpreted as a kind of physical or functional product description [18], which is used as a basic framework for formulating capabilities of suppliers and requirements of customers. Figure 3 represents fragments of an

[7] See http://www.ilog.com for reference.

ontology for defining configuration services for IP-based Virtual Private Networks (IP-VPN)[8]. For our example we will concentrate on

```
begin-ontology
ontology-container
title Product Domain Ontology
description  IP-based Virtual Private Networks

language  "OIL"
ontology-definitions
slot-def protocol
subslot-of
  HasPart    //defined in Configuration Ontology
class-def AccessProtocol
subclass-of
  Function   //defined in Configuration Ontology

class-def RouterAccess
subclass-of
  AccessProtocol
class-def ModemAccess
subclass-of
  AccessProtocol
class-def InternetAccess
subclass-of
  AccessProtocol
class-def defined  Country

class-def defined  Town
  slot-constraint  town_of  cardinality  1 Country
class-def LineService
subclass-of
  Function   //defined in Configuration Ontology
  slot-constraint  bandwidth  cardinality  1 integer
  slot-constraint  latency  cardinality  1 integer
  slot-constraint  identifier  has-value  integer
class-def BackBoneSection
subclass-of
  LineService
  slot-constraint  access_from  has-value  AccessLine

class-def defined  AccessLine
subclass-of
  LineService
  slot-constraint  protocol  cardinality  1 AccessProt ocol
  slot-constraint  access_to  cardinality  1 BackBoneSection
  slot-constraint  pop  cardinality  1 Town
instance-of UK Country

instance-of Manchester  Town
related town_of  Manchester UK
end-ontology
```

Figure 3. Domain ontology for IP-VPN services

the provision of *AccessLines* that connect a customer location (slot *pop* - 'point of presence') to a *BackBoneSection*. The chosen *protocol* (a refinement of the *HasPart* decomposition relationship in the configuration domain) can be either performed via a router, via a modem or via an internet connection to some access gateway (*RouterAccess*, *ModemAccess* and *InternetAccess* are therefore specialized *AccessProtocols*). In addition, an *AccessLine* is characterized by a *bandwidth* and *latency* property that it inherits from its superclass *LineService*, which is in turn a refinement of the *Function* concept (abstract characteristic of a product or service) from the basic configuration ontology (*intermediate ontology level*). The instances contained in the ontology shown in Figure 3 can be interpreted as basic catalog entries representing common knowledge (e.g., British towns or zip codes), which are assigned to base classes of the application domain ontology (in this case *Manchester* is provided as basic instance of the class *Town*).

[8] The complete example ontology in DAML+OIL can be downloaded from http://www.cawicoms.org/ontology/ipvpn.rdfs.

5 Web service scenario

The interaction between service providing agents can be differentiated into the three areas service publishing, identification and execution. As depicted in the scenario in Figure 2, only those agents can request a service that have the mediating capabilities to receive service advertisements and perform service identification.

5.1 Service publishing

Now we will show how the ontologies defined in Section 4 are used to semantically describe the offered configuration services. Semantic description of the demanded services allows us to implement efficient matchmaking between supply and demand. Within these semantic annotations, restrictions on the domain and cardinality of slots, constraints on connections and structure, as well as the possible types of classes are possible. Furthermore, offered component instances can be represented as subconcepts (e.g. read from a catalog) of the classes of the service domain-specific configuration ontology. Additional supplier-specific constraints are introduced. Consequently, for the semantic description of the capability of a configuration service of a specific supplier the product domain ontology level provides the necessary base concepts that can be further refined. Figure 4 contains the semantic definition of the *AccessLine* services that are offered by the fictitious telecommunication service providers *BTT* and *Luton*. *BTT* serves customers located in the UK and Ireland (constraint on the slot pop) and can provide access to *BackBoneSections* 1 through 10 with a maximum *bandwidth* of 2000. In contrast *Luton* offers connections from towns in France and the UK. Only modem or internet are offered protocol choices, a lower *bandwidth* is supported and fewer *BackBoneSections* are accessible. For tailoring the application

```
class-def defined  BTT_AccessLine
subclass-of
  AccessLine
  slot-constraint  access_to  value-type
    ((slot-constraint  identifier  value-type (min  1)) and
    (slot-constraint  identifier  value-type (max  10)))
  slot-constraint  pop  value-type ((slot-constraint  town_of  value-type (one-of UK))
    or (slot-constraint  town_of  value-type (one-of Ireland)))
  slot-constraint  bandwidth  value-type (max  2000)

class-def defined  Luton_AccessLine
subclass-of
  AccessLine
  slot-constraint  pop  value-type ((slot-constraint  town_of  value-type (one-of France))
    or (slot-constraint  town_of  value-type (one-of UK)))
  slot-constraint  access_to  value-type
    ((slot-constraint  identifier  value-type (min  5)) and
    (slot-constraint  identifier  value-type (max  8)))
  slot-constraint  bandwidth  value-type (max  1200)
  slot-constraint  protocol  has-value (ModemAccess  or  InternetAccess )
```

Figure 4. Semantic description of offered services

domain specific configuration ontology to supplier-specific circumstances tool support for acquisition and maintenance of configuration models is needed. Within the CAWICOMS project a *Knowledge Acquisition Workbench* is developed that provides the required tools for designing the service descriptions with a graphical UML-based notation. The generic and the intermediate ontology level as described in Section 4 are inherent to the modeling primitives offered by the tool suite and therefore static in our approach. The tool environment supports human experts in defining and maintaining the application domain specific ontological descriptions as well as in integrating them. The advertisement of the offered configuration services of different suppliers is therefore part of an offline setup process. The functional descriptions of the configurable products and services are communicated to all Web configurators that may act as customers for their configuration service and integrated into their domain ontologies by the human experts.

5.2 Service Identification

Having described service publication, we will now focus on the identification of relevant Web service providers for a concrete demand. This task has similarities with the *surgical* or *parametric search* problem [16], e.g. *"a laptop with at least 20GB hard-disk, 800MHz Pentium III processor or better, manufactured either by Dell or Compaq and costing less than 2000 USD"*. However, for the configuration domain we require even more enhanced search capabilities for identifying the appropriate supply. The reason is, that requirements cannot only be expressed as simple restrictions on product attributes, but also as constraints on the structure. The following example is based on the product domain ontology (Figure 3), requestors are enabled to semantically describe the requested service as can be seen in Figure 5. Let us assume that we search for an *AccessLine* provider that connects us from *Manchester* via *InternetAccess* protocol to *BackBoneSection* '3' with a *bandwidth* of 1200. Here the *bandwidth* slot-constraint is a simple attribute restriction, but the constraint on the slot *access_to* navigates to the related class *BackBoneSection* and restricts the structure. For this

```
class-def defined  Required_AccessLine
subclass-of
  AccessLine
  slot-constraint  bandwidth value-type (equal 1200)
  slot-constraint  pop value-type (one-of Manchester )
  slot-constraint  protocol value-type InternetAccess
  slot-constraint  access_to has-value
    (slot-constraint  identifier  has-value (equal 3))
```

Figure 5. Semantic description of required service

example we can intuitively determine that *BTT* is an appropriate supplier for the requested service, as the *Required_AccessLine* qualifies as a subclass to *BTT_AccessLine*. However, for the general case identification of subsumption relationships between offered and required concepts is too restrictive. Consider the case where we would need this *AccessLine* either from *Manchester* or from *Munich*. Assuming all other restrictions remain unchanged, the modified constraint on the slot pop is given in Figure 6[9]. Although *BTT* still provides an appropriate service, the constraint relaxation makes the subsumption of *Required_AccessLine* by *BTT_AccessLine* impossible. So formally a description logic definition of the matchmaking task for identification of an appropriate configuration service can be defined as follows.

Given: *A consistent description logic theory T that represents the three ontological layers of our marketplace, a set of concepts $S = \{S_1, \ldots, S_n\}$ that describe supply from n different suppliers (i.e., service providers), and a set of concepts D representing the requested service.*

Task: *Identify the set of concepts A, that contains all concepts S_a with $S_a \in A$, where S_a is an appropriate service for D*

[9] Note, that the inherited cardinality constraint restricts slot *pop* to exactly one *Town*, which gives this constraint an *exclusive or* semantics.

```
class-def defined Required_AccessLine_1
subclass-of
  AccessLine

  slot-constraint pop value-type ((one-of Manchester) or (one-of Munich))
```

Figure 6. Modified service requirement

and $A \subseteq S$.

Definition (appropriate service): *A service S_a is an appropriate service for D, iff $S_a \cup D$ are consistent.*

Note, that this definition diverges from the approaches taken for matchmaking among heterogenous agents [27] or for Web service identification [17]. For a detailed elaboration on the relationship between description logic and configuration knowledge representation see [6].
As already mentioned in the previous subsection, the configuration service models are defined within a knowledge acquisition environment and automatically translated into the proprietary knowledge representation formalism of a configuration agent. In our implementation this matchmaking task is therefore performed as part of the search process for a configuration solution of a constraint-based configurator engine. For the internal representation of the advertised service models as well as the service requests an object-oriented framework for constraint variables is employed [14]. Reasoning on service requirements as well as on service decomposition is performed by the underlying Java-based constraint solver. The formulation of service requests and their replies is enabled by a *WebConnector* component that owns an object model layer that accesses the internal constraint representation of the constraint engine. This object model layer represents the advertised configuration service descriptions. A service requestor agent imposes its service request via an *edit-query* onto the object-model layer and retrieves the configuration service result via a *publish-query*.
As will be also pointed out in the next subsection, the creation of standards for the definition of semantics of Web services will allow application independent mediating agents to accept service advertisements and to perform the service identification task, which is not the case in the current situation.

5.3 Service Execution

Requests for service execution must conform to an XML-based communication protocol (*WebConnector* protocol) developed for the configuration domain in accordance with the SOAP messaging standard. This protocol defines

- a fixed set of methods with defined semantics for the configuration domain, like creating components, setting values for parameters, initiation of the search process, or retrieving results,
- a mechanism to exchange complex data structures like configuration results and a language for expressing navigation expressions within these data structures (compare to XML-Schema and XPath), and
- extensibility mechanisms for special domains and support for a session concept in HTTP-based transactions.

This way the semantics of the process model of the configuration Web service is defined by a proprietary protocol. This assumption works for our specific requirement of realizing collaborative configuration systems, but is only half way towards the vision of Web services in the Semantic Web. Therefore, markup languages are required that enable a standardized representation of service profiles for advertisement of services as well as definitions of the process model. This way, the task of identifying appropriate services and the decomposition of a service request into several separate requests can be performed by domain independent mediators. Due to the lack of these standards, this mediating functionality is in our case performed by application logic integrated into the configuration systems. DAML-S[10] is an example for an effort underway that aims at providing such a standardized semantic markup for Web services that builds on top of DAML+OIL.

6 Related Work

Beside standards for representing product catalogs [8], there exists a number of approaches for standardizing electronic commerce communication (e.g. Commerce XML - cXML or Common Business Library - CBL) - these are XML-based communication standards for B2B applications[11], which also include basic mechanisms for product data interchange and can be interpreted as ontologies supporting standardized communication between e-Business applications. However, these standards are restricted to the representation of standardized products, i.e. the basic properties of complex products, especially configurable products are not considered. Basic mechanisms for product data integration are already supported by a number of state-of-the-art B2B applications. However, the integration of configuration systems into electronic marketplace environments is still an open issue, i.e. not supported by todays systems. Problem Solving Methods (PSMs) [2] support the decomposition of reasoning tasks of knowledge-based systems into sets of subtasks and inference actions that are interconnected by knowledge roles. The goal of the IBROW project [20] is the semiautomatic reuse of available problem solving methods, where a software broker supports the knowledge engineer in configuring a reasoning system by combining different PSMs. A similarity to the work of [20] exists in the sense that the selection of suppliers (and corresponding configuration systems) is a basic configuration task, where configurators must be selected which are capable of cooperatively solving a distributed configuration task. The approach is different in the sense that the major focus is on providing an environment which generally supports a semi-automated reuse of problem solving methods, whereas our approach concentrates on the automated integration of configuration services in an e-business environment. The Infomaster system [15] provides basic mechanisms for integrating heterogeneous information sources in order to provide a unique entry point for the users of the system. Compared to our approach there is no support for the integration of configurable products and the underlying configuration systems. The design of large scale products requires the cooperation of a number of different experts. In the SHADE (Shared Dependency Engineering) project [22] a KIF-based representation [21] was used for representing engineering ontologies. This approach differs from the approach presented in this paper in the sense that the provided ontology is majorly employed as a basis for the communication between the different engaged agents, but is not used as a means for describing the capabilities of agents. The STEP standard [13] takes into account all aspects of a product including geometry and organisational data [19]. The idea of STEP is to provide means for defining application specific concepts for modeling products in a particular application domain. These application specific concepts are standardised into parts of STEP called *Application Protocols* which are defined using the EXPRESS data definition language (Application Protocols are EXPRESS schemas). EXPRESS itself includes a set of modeling concepts useful for representing configurable products, however the language can not be used to define an enterprise specific configuration model without leaving the STEP standard. Similarities to our approach can be seen in the role of application protocols in STEP which are very similar to the domain ontology level discussed in this paper.

[10] See http://www.daml.org/services for reference.

[11] An overview on existing e-Commerce frameworks for business to business communication can be found in [25].

7 Conclusions

The Semantic Web [3] is the vision of developing enabling technologies for the Web which supports access to its resources not only to humans but as well to applications often denoted as agent-based systems providing services such as information brokering, information filtering, intelligent search or synthesis of services [20]. This paper describes an application scenario for semantic Web services in the domain of configuring telecommunication services. It demonstrates how to apply Semantic Web technologies in order to support the integration of configurable products and services in an environment for distributed problem solving. DAML+OIL-based configuration service descriptions can be used in order to match them with given customer requirements and the matchmaking task to determine the adequacy of a service is defined. DAML+OIL formalisms are well suited for representing the component structure of configurable products, i.e. part-of associations and simple associations between component types and corresponding basic constraints. However, technologies supporting the vision of the Semantic Web are still under development. In order to support a full scenario of distributed configuration Web services, languages like DAML+OIL have to be extended with language elements supporting the formulation of service advertisements as well as process definitions for the interaction.

REFERENCES

[1] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens, 'OilEd: A Reasonable Ontology Editor for the Semantic Web', in *Proceedings of Joint Austrian/German Conference on Artificial Intelligence (KI)*, pp. 396–408, Vienna, Austria, (2001).

[2] R. Benjamins and D. Fensel, '', *Special issue on problem-solving methods of the International Journal of Human-Computer Studies*, **49**(4), (1998).

[3] T. Berners-Lee, *Weaving the Web*, Harper Business, 2000.

[4] B. Chandrasekaran, J. Josephson, and R. Benjamins, 'What Are Ontologies, and Why do we Need Them?', *IEEE Intelligent Systems*, **14,1**, 20–26, (1999).

[5] A. Felfernig, G. Friedrich, and D. Jannach, 'UML as domain specific language for the construction of knowledge-based configuration systems', *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, **10**(4), 449–469, (2000).

[6] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker, 'A Joint Foundation for Configuration in the Semantic Web', *In Proceedings of the Workshop on Configuration, in conjunction with the 15th European Conference on Artificial Intelligence (ECAI-2002)*, (2002).

[7] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, 'Web-based configuration of Virtual Private Networks with Multiple Suppliers', in *Proceedings of the 7th International Conference on Artificial Intelligence in Design (AID)*, Cambridge, UK, (2002).

[8] D. Fensel, Y. Ding, B. Omelayenko, E. Schulten, G. Botquin, M. Brown, and A. Flett, 'Product Data Integration in B2B E-Commerce', *IEEE Intelligent Systems*, **16**(4), 54–59, (2001).

[9] D. Fensel, F. vanHarmelen, I. Horrocks, D. McGuinness, and P.F. Patel-Schneider, 'OIL: An Ontology Infrastructure for the Semantic Web', *IEEE Intelligent Systems*, **16**(2), 38–45, (2001).

[10] A. Gangemi, D. M. Pisanelli, and G. Steve, 'An Overview of the ONIONS Project: Applying Ontologies to the Integration of Medical Terminologies', *Data and Knowledge Engineering*, **31**(2), 183–220, (1999).

[11] T. Gruber, 'A translation approach to portable ontology specifi cations', *Knowledge Acquisition*, **5**, 199–220, (1993).

[12] J. Hendler, 'Agents and the Semantic Web', *IEEE Intelligent Systems*, **16**(2), 30–37, (2001).

[13] ISO, 'ISO Standard 10303-1: Industrial automation systems and integration - Product data representation and exchange - Part 1: Overview and fundamental principles', (1994).

[14] U. Junker, 'Preference-programming for Confi guration', in *Proceedings of IJCAI, Confi guration Workshop*, Seattle, (2001).

[15] A. M. Keller and M. R. Genesereth, 'Multivendor Catalogs: Smart Catalogs and Virtual Catalogs', *The Journal of Electronic Commerce*, **9**(3), (1996).

[16] D.L. McGuinness, 'Ontologies and Online Commerce', *IEEE Intelligent Systems*, **16**(2), 9–10, (2001).

[17] Sh. McIlraith, T.C. Son, and H. Zeng, 'Mobilizing the Semantic Web with DAML-Enabled Web Services', in *Proceedings of the IJCAI 2001 Workshop on E-Business and the Intelligent Web*, pp. 29–39, Seattle, WA, (2001).

[18] S. Mittal and F. Frayman, 'Towards a Generic Model of Confi guration Tasks', in *Proceedings* 11^{th} *International Joint Conf. on Artifi cial Intelligence*, pp. 1395–1401, Detroit, MI, (1989).

[19] T. Mnnist, A. Martio, and R. Sulonen, 'Modelling generic product structures in STEP', *Computer-Aided Design*, **30,14**, 1111–1118, (1999).

[20] E. Motta, D. Fensel, M. Gaspari, and V.R. Benjamins, 'Specifi cations of Knowledge Components for Reuse', in *Proceedings of* 11^{th} *International Conference on Software Engineering and Knowledge Engineering*, pp. 36–43, Kaiserslautern, Germany, (1999).

[21] 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).

[22] G.R. Olsen, M. Cutkosky, J.M. Tenenbaum, and T.R. Gruber, 'Collaborative Engineering based on Knowledge Sharing Agreements', in *Proceedings of the ACME Database Symposium*, pp. 11–14, Minneapolis, MN, USA, (1994).

[23] B.J. PineII, B. Victor, and A.C. Boynton, 'Making Mass Customization Work', *Harvard Business Review*, **Sep./Oct. 1993**, 109–119, (1993).

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

[25] S.S.Y. Shim, V.S. Pendyala, M. Sundaram, and J.Z. Gao, 'E-Commerce Frameworks', *IEEE Computer*, **Oct. 2000**, 40–47, (2000).

[26] T. Soininen, J. Tiihonen, T. Mnnist, and R. Sulonen, 'Towards a General Ontology of Confi guration', *AI Engineering Design Analysis and Manufacturing Journal, Special Issue: Confi guration Design*, **12**(4), 357–372, (1998).

[27] K. Sycara, M. Klusch, and S. Widoff, 'Dynamic Service Matchmaking among Agents in Open Information Environments', *ACM SIGMOD Record, Special Issue on Semantic Interoperability in Global Information Systems*, (1999).

A Joint Foundation for Configuration in the Semantic Web

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

Abstract. Product configuration is a major commercial application of knowledge-based systems, and joint configuration by multiple business partners is becoming a key application in today's highly specialized economy. The required integration of configuration knowledge is a challenging task due to the variety of knowledge representation formalisms used in commercial configurators. Ontology languages such as DAML+OIL provide an infrastructure for the Semantic Web with the goal of intelligent information integration. The aim of this paper is to show the applicability of such languages for building configuration knowledge bases. We join the two major streams in knowledge-based configuration (description logics on one hand and predicate logic, including constraint-based and resource-balancing techniques on the other) by giving a description logic based definition of a configuration task and showing its equivalence with existing consistency-based definitions. We show that Semantic Web ontology languages can be applied to configuration by formalizing language elements relevant for building configuration knowledge bases and discuss extensions needed in order to provide full fledged configuration knowledge representation. The result is a common basis for current configuration approaches on the Semantic Web that is necessary for the provision of joint configuration services.

1 Introduction

Knowledge-based configuration has a long history as a successful AI application area (e.g., [3, 10, 12, 13, 15, 19]). Starting with rule-based systems such as R1/XCON [3], various higher level representation formalisms were developed to exploit the advantages of more concise representation, faster application development, higher maintainability and more flexible reasoning. Although these representations have proven their applicability in various real world applications, the heterogeneity of configuration knowledge representation is the major obstacle to incorporating configuration technology in eCommerce environments. The trend towards highly specialized solution providers results in a situation where different configurators of complex products and services must be integrated in order to transparently support distributed configuration problem solving.

For this integration, configurators must have a clear common understanding of the problem definition and the semantics of the exchanged knowledge. Consequently, it is necessary to agree on the definition of a configuration problem and its solution. Of the two current main streams in representing and solving configuration problems, the first approach is based on predicate logic or various simplified variants thereof, specifically constraint-based systems (including their dynamic and generative variants, e.g., [10, 13, 14]) and resource balancing methods (e.g., [12]). The second approach uses description logics as knowledge representation and reasoning mechanism (e.g., [19]). Clearly, an integration of these approaches is a major milestone for configurator integration.

A solution for the exchange of knowledge is the provision of a standardized configuration knowledge representation language which is based on state-of-the-art Web technologies allowing easy integration of existing proprietary configuration environments. Ontology representation languages such as OIL [9] or DAML+OIL [18] developed in the context of the Semantic Web [4] are well suited for designing and sharing ontologies. These languages are strongly influenced by description logics and therefore possess clear declarative semantics, providing one important precondition for the exchange of knowledge. Nonetheless, their roots in description logics reinforces the need for the definition of a common view of a configuration task, so that predicate logic based representations can be mapped to them.

The practical consequence of a commonly accepted problem definition and knowledge semantics in joint provisioning of configuration services is a well defined interface between configurator implementations. Proprietary configuration systems can be independently implemented following different approaches and are still able to interoperate. We only require that cooperating configurators deliver valid solutions w.r.t. the common definition of the problem, the solution, and the semantics of the exchanged knowledge.

In this paper we give a description logics based definition of a configuration task and show the equivalence of this definition with a consistency-based definition given in [8] - the major result of this equivalence is that configuration tasks defined in terms of description logics and predicate logic can easily be transformed into each other and consequently be represented in ontology representation languages such as DAML+OIL. Using concepts of OIL[3], we present the constituting elements of a configuration knowledge representation language by formalizing modeling concepts of de facto standard configuration ontologies [7, 17] employed in industrial applications[4]. In addition, we point out extensions needed to apply OIL and DAML+OIL for full fledged configuration knowledge representation. As a result, we provide a common basis for knowledge representation in configuration problem solving, thus enhancing the applicability of configuration technology to Web-based environments.

The rest of the paper is organized as follows. In Section 2 we intro-

[1] Institut für Wirtschaftsinformatik und Anwendungssysteme, Produktionsinformatik, Universitätsstrasse 65-67, A-9020 Klagenfurt, Austria, email: {*felfernig,friedrich,jannach,zanker*}@ifit.uni-klu.ac.at.
[2] University of South Australia, Advanced Computing Research Centre, 5095 Mawson Lakes (Adelaide), SA, Australia, *email: mst@cs.unisa.edu.au.*

[3] For presentation purposes we employ OIL text - this representation can easily be transformed into a corresponding DAML+OIL representation.

[4] Note that these differ somewhat from the configuration ontology that was described for demonstration purposes in [11].

duce an example that provides an overview of the modeling concepts required for building configuration knowledge bases. In Section 3 we give a description logics based definition of a configuration task and show its equivalence to the consistency-based definition given in [8]. In Section 4 we describe an OIL-based formalization of the modeling concepts presented in Section 2 and summarize the results. Section 5 closes with conclusions.

2 Configuration domain specific modeling concepts

In the following we discuss a set of relevant modeling concepts for building configuration knowledge bases. These concepts are extracted from the configuration ontologies defined in [7, 17]. Figure 1 shows the simplified structure of a configurable *Computer* system which is composed of the following representation concepts.

Component types represent the parts, a final product is built of - they are characterized by attributes (e.g., the component type *CPU* is characterized by the attribute *clockrate*). Component types with a similar structure are arranged in a *generalization* hierarchy and represent choices for the configurable product (e.g., in the final configuration an instance of *CPU* can be either a *CPU1* or a *CPU2*). *Part-whole* relationships can be considered as bill-of-material representations semantically enriched with multiplicities, stating a range of how many subparts an aggregate can consist of (e.g., a *MB* must contain at least one *CPU* and at most two *CPUs*). In addition to the number and types of different components, the product topology may be important in a final configuration as well, i.e., how the components are interconnected to each other (e.g., which *videoport* is used in the configuration to connect the *Videocard* with the *Screen*).

Additionally, a set of constraints specifies allowed combinations of component- and attribute settings in the final configuration. Some component types cannot be used in the same final configuration (they are *incompatible*). E.g., we can impose the constraint on the product structure of Figure 1 that a *CPU1* is incompatible with a motherboard *MB2*. In some cases, the existence of a component of a certain type *requires* the existence of an instance of another specific type. Regarding the product structure of Figure 1, we can impose the constraint that the existence of a *CPU2* requires the existence of a *MB*2. Finally, parts of a configuration problem can be interpreted as a resource balancing task, where some of the components are *producers* and others are *consumers*. In the final configuration, consumers and producers must be balanced w.r.t. some resource balancing criteria - e.g., the amount of installed hard-disk capacity is the upper bound for the capacity requirements of the installed software. In Section 4 we will show how to represent these concepts in extended OIL [9].

3 Defining configuration tasks

For the description of a configuration task we employ a description logic language (e.g., OIL) starting from a schema $S = (\mathcal{CN}, \mathcal{RN}, \mathcal{IN})$ of disjoint sets of names for concepts, roles, and individuals [5]. Concepts can be seen as unary predicates defining classes (component types). Roles are used to express relationships between different elements of a domain. Finally, individuals are specific named elements of the domain[5].

Definition 1 (Configuration problem in description logic): *In general we assume a configuration problem is described by a triple* $(DD_{DL}, SRS_{DL}, CLANG_{DL})$. DD_{DL} *represents the domain description of the configurable product and* SRS_{DL} *specifies the particular system requirements defining an individual configuration problem instance.* $CLANG_{DL}$ *comprises a set of concepts* $C_{config} \subseteq \mathcal{CN}$ *and a set of roles* $R_{config} \subseteq \mathcal{RN}$ *which serve as a configuration language for the description of actual configurations (solutions). A configuration knowledge base* $KB_{DL} = DD_{DL} \cup SRS_{DL}$ *is constituted of sentences in a description language.* □

[5] In the following we assume that the reader is familiar with the concepts of OIL. See [9] for an introductory text.

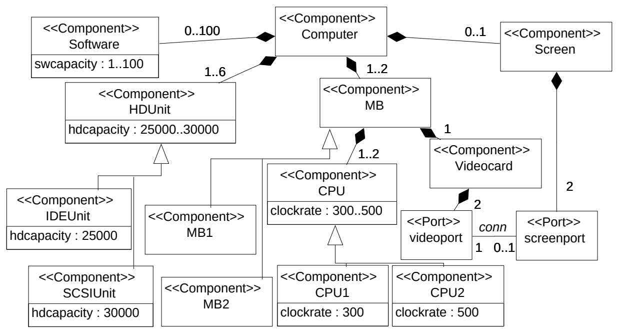


Figure 1. Example configuration model

In the following we will define a solution of a configuration problem based on the interpretation of concepts and roles. In addition we require that roles in $CLANG_{DL}$ are defined over the domains given in C_{config}, i.e., we add for each $R_i \in R_{config}$ the role descriptions $range(R_i) = CDom$ and $dom(R_i) = CDom$ for $CDom \doteq \bigsqcup_{C_i \in C_{config}} C_i$ to the knowledge base KB_{DL} if such descriptions are not subsumed by other descriptions already contained in the knowledge base.

Example 1: In this example we use a part of our *Computer* ontology (see Figure 1) that comprises *CPUs* and *MBs*. On each *MB* at least one but at most two *CPUs* are plugged in (constraint c_1). A *CPU* must always be mounted on a *MB* (constraint c_2). A *CPU* of type *CPU2* must be mounted on a *MB* of type *MB2* (constraint c_3). The domain description DD_{DL}={

class-def MB subclass-of ($MB1$ or $MB2$)
  slot-constraint *cpu-of-mb* min-cardinality 1 CPU
  slot-constraint *cpu-of-mb* max-cardinality 2 CPU. **[**c_1**]**
class-def $MB1$ subclass-of MB.
class-def $MB2$ subclass-of MB.
disjoint $MB1$ $MB2$.
class-def CPU subclass-of ($CPU1$ or $CPU2$)
  slot-constraint *mb-of-cpu* cardinality 1 MB. **[**c_2**]**
class-def $CPU1$ subclass-of CPU.
class-def $CPU2$ subclass-of CPU
  slot-constraint *mb-of-cpu* cardinality 1 $MB2$. **[**c_3**]**
disjoint $CPU1$ $CPU2$.
disjoint CPU MB.
slot-def *mb-of-cpu*
  inverse *cpu-of-mb* domain CPU range MB.
slot-def *cpu-of-mb*
  inverse *mb-of-cpu* domain MB range CPU.} □

The customer requirement "two CPUs of type *CPU1* and one CPU of type *CPU2"* is expressed by $SRS_{DL} = \{$ (*instance-of c1 CPU1*), (*instance-of c2 CPU1*), (*instance-of c3 CPU2*) $\}$. The configuration language $CLANG_{DL}$ is defined by $C_{config} = \{CPU1, CPU2, MB1, MB2\}$ and $R_{config} = \{$*mb-of-cpu*$\}$.

In our example we do not include the concepts CPU and MB and the role *cpu-of-mb* in $CLANG_{DL}$ since we are only interested in the leaf concepts of a generalization hierarchy and specific relationships (e.g., to manufacture the final system we only need to know the most specific type for each component and its connections).

The semantics of description terms are usually given denotationally using an interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, (\cdot)^{\mathcal{I}} \rangle$, where $\Delta^{\mathcal{I}}$ is a domain (non-empty universe) of values, and $(\cdot)^{\mathcal{I}}$ a mapping from concept descriptions to subsets of the domain, and from role descriptions to sets of 2-tuples over the domain. The mapping also associates with every individual name in $\mathcal{IN}$ some distinct value in $\Delta^{\mathcal{I}}$. The reason for this distinctness is the unique name assumption (UNA) we employ in our formalism. We require the UNA for concepts and roles which describe configurations in order to make sure that different identifiers for individuals (e.g., modules of a system) refer to different individuals. This UNA can be lifted if necessary for concepts and roles which are not used to describe configurations. In the following we give a description logic based definition of a configuration problem and show its equivalence with consistency-based definitions given in the literature [8]. This definition serves as a joint foundation of configuration knowledge representation in the Semantic Web.

Definition 2 (Valid configuration in description logic): *Let $\mathcal{I} = \langle \Delta^{\mathcal{I}}, (\cdot)^{\mathcal{I}} \rangle$ be a model of a configuration knowledge base KB_{DL}, $CLANG_{DL} = C_{config} \cup R_{config}$ a configuration language, and $CONF_{DL} = COMPS \cup ROLES$ a description of a configuration. $COMPS$ is a set of tuples $\langle C_i, INDIVS_{C_i} \rangle$ for every $C_i \in C_{config}$, where $INDIVS_{C_i} = \{ci_1, \ldots, ci_{n_i}\} = C_i^{\mathcal{I}}$ is the set of individuals of concept C_i. These individuals identify components in an actual configuration. $ROLES$ is a set of tuples $\langle R_j, TUPLES_{R_j} \rangle$ for every $R_j \in R_{config}$ where $TUPLES_{R_j} = \{\langle rj_1, sj_1 \rangle, \ldots, \langle rj_{m_j}, sj_{m_j} \rangle\} = R_j^{\mathcal{I}}$ is the set of tuples of role R_j defining the relation of components in an actual configuration.* □

Example 2: A valid configuration for our example knowledge base is $CONF_{DL} = \{\langle CPU1, \{c1, c2\}\rangle, \langle CPU2, \{c3\}\rangle, \langle MB1, \{m1\}\rangle, \langle MB2, \{m2\}\rangle, \langle \textit{mb-of-cpu}, \{\langle c1, m1\rangle, \langle c2, m1\rangle \langle c3, m2\rangle\}\rangle\}$. □

We also have to describe component parameter settings in addition to components and their connections. Using description logics, parameter settings of components are modeled by special functional roles (also called *features*) expressing the relation between the component and the data value assigned to a particular attribute. Therefore, component structure and parameter settings can be treated in a uniform manner except that the parameter values come from some data value domain $dom(D)$ [16] disjunct from the individuals in $COMPS$.

In addition to the definition of a valid configuration given above, we can provide an equivalent characterization based on checking the consistency of a set of axioms.

Remark 1: Let KB_{DL} be a configuration knowledge base, $CLANG_{DL} = C_{config} \cup R_{config}$ a configuration language, and $CONF_{DL} = COMPS \cup ROLES$ a description of a configuration.

The concepts C_i are defined by the component axioms $AX_{COMPS} =$

$$\{C_i \doteq \text{one-of}[ci_1, \ldots, ci_{n_i}] | \langle C_i, INDIVS_{C_i}\rangle \in COMPS \text{ where}$$

$$INDIVS_{C_i} = \{ci_1, \ldots, ci_{n_i}\}\}.$$

The roles R_j are defined by the role axioms $AX_{ROLES} =$

$$\{R_j \doteq \bigsqcup_{\langle rj, sj\rangle \in TUPLES_{R_j}} \text{product}[\text{one-of}[rj], \text{one-of}[sj]] |$$

$$\langle R_j, TUPLES_{R_j}\rangle \in ROLES \text{ with}$$

$$TUPLES_{R_j} = \{\langle rj_1, sj_1\rangle, \ldots, \langle rj_{m_j}, sj_{m_j}\rangle\}\}.$$

$CONF_{DL}$ is a valid configuration *iff* $KB_{DL} \cup AX_{COMPS} \cup AX_{ROLES}$ is satisfiable.□

Note that we use the above notation for defining the complete extension of roles in order to apply the translation function from description logics to predicate logic defined in [5]. An alternative to the *product* constructor would be a constructor that (in a similar manner to the *one-of* constructor for concepts) permits the enumeration of permissible entries for a role. However, the usual complexity problems with the *product* constructor do not actually arise since we only apply *product* to singleton arguments.

Example 3: In order to verify that a given configuration is valid w.r.t. our example KB_{DL}, we need to add the axioms $CPU1 \doteq$ one-of$[c1, c2]$, $CPU2 \doteq$ one-of$[c3]$, $MB1 \doteq$ one-of$[m1]$, $MB2 \doteq$ *one-of*$[m2]$, *mb-of-cpu* $\doteq$ *product*[*one-of*$[c1]$, *one-of*$[m1]$] $\sqcup$ *product*[*one-of*$[c2]$, *one-of*$[m1]$]$\sqcup$ *product*[*one-of*$[c3]$, *one-of*$[m2]$].□

We now give a consistency-based definition of a configuration problem using predicate logic (this definition corresponds to the definition given in [8]) and show the equivalence with the description logic based definition given before.

Definition 3 (Configuration problem in predicate logic): *In general we assume a configuration problem is described by a triple $(DD_{LOG}, SRS_{LOG}, CLANG_{LOG})$ where DD_{LOG} and SRS_{LOG} are logical sentences and $CLANG_{LOG}$ is a set of predicate symbols. DD_{LOG} represents the domain description and SRS_{LOG} specifies the particular system requirements. A configuration $CONF_{LOG}$ is described by a set of positive ground literals whose predicate symbols are in the set of $CLANG_{LOG}$.* □

Example 4: For our example, DD_{LOG} can be expressed by using monadic and dyadic predicates and numerical quantifiers, i.e.,
$DD_{LOG} = \{$
$\forall X : MB(X) \leftrightarrow MB1(X) \vee MB2(X).$
$\forall X : \neg MB1(X) \vee \neg MB2(X).$
$\forall X : MB(X) \rightarrow \exists_1^2 Y : \textit{mb-of-cpu}(X, Y).$
$\forall X : CPU(X) \leftrightarrow CPU1(X) \vee CPU2(X).$
$\forall X : \neg CPU1(X) \vee \neg CPU2(X).$
$\forall X : CPU(X) \rightarrow \exists_1^1 Y : \textit{mb-of-cpu}(Y, X).$
$\forall X : CPU2(X) \rightarrow \exists_1^1 Y : \textit{mb-of-cpu}(Y, X) \wedge MB2(Y).$
$\forall X : \neg MB(X) \vee \neg CPU(X).$
$\forall X, Y : \textit{mb-of-cpu}(X, Y) \leftrightarrow \textit{cpu-of-mb}(Y, X).$
$\forall X, Y : \textit{mb-of-cpu}(X, Y) \rightarrow \textit{mb}(X) \wedge \textit{cpu}(Y)\}.$
$SRS_{LOG} = \{CPU1(c1).\ CPU1(c2).\ CPU2(c3).\}.$
$CLANG_{LOG} = \{CPU1, CPU2, MB1,$
$MB2, \textit{mb-of-cpu}\}$.□

Definition 4 (Consistent configuration in predicate logic): *Given a configuration problem $(DD_{LOG}, SRS_{LOG}, CLANG_{LOG})$, a configuration $CONF_{LOG}$ is consistent iff $DD_{LOG} \cup SRS_{LOG} \cup CONF_{LOG}$ is satisfiable.* □

This definition allows determining the consistency of partial configurations, but does not guarantee the completeness of configurations [8]. It is necessary that a configuration explicitly includes all needed components (and their connections and attribute values), in order to assemble a correctly functioning system. We need to introduce an explicit formula for each predicate symbol in $CLANG_{LOG}$ to enforce its completeness property. In order to stay within first order logic, we model the property by first order formulae. For our example we have to add the completeness axiom $\forall X : CPU1(X) \Rightarrow (\bigvee_{Z \in CONF_{LOG}} CPU1(X) = Z)$ for the predicate $CPU1$ and similar axioms for $CPU2$ and *mb-of-cpu*.

Note that $\bigvee_{Z \in CONF_{LOG}} CPU1(X) = Z$ serves as a macro which is expanded based on the elements in $CONF_{LOG}$. We refer to $CONF_{LOG}$ extended by completeness axioms as $\widehat{CONF}_{LOG}$.

Example 5: $CONF_{LOG} = \{CPU1(c1).\ CPU1(c2).\ CPU2(c3).\ MB1(m1).\ MB2(m2).$ *mb-of-cpu*$(c1, m1).$ *mb-of-cpu*$(c2, m1).$ *mb-of-cpu*$(c3, m2).\}$

The completeness axiom for $CPU1$ is $\forall X : CPU1(X) \Rightarrow CPU1(X) = CPU1(c1) \vee CPU1(X) = CPU1(c2)$, where unsatisfiable literals are deleted. □

Definition 5 (Valid configuration in predicate logic): *Let* $(DD_{LOG}, SRS_{LOG}, CLANG_{LOG})$ *be a configuration problem. A configuration* $CONF_{LOG}$ *is valid iff* $DD_{LOG} \cup SRS_{LOG} \cup \widehat{CONF}_{LOG}$ *is satisfiable.* □

Note that $CONF_{LOG}$ in Example 5 is a valid configuration.

In order to show the equivalence of valid configurations for description logic and predicate logic we apply a translation function $\mathcal{T}\langle\cdot\rangle$ that maps description logics to predicate logic, i.e., axioms to formulas with no free variables, concepts to formulas with one free variable x, and roles to formulas with two free variables x and y.

Borgida [5] provides such a translation function $\mathcal{T}\langle\cdot\rangle$ such that concepts, roles, terms, and axioms are translated into equivalent formulas in predicate logic. $KB_{LOG} = \mathcal{T}\langle KB_{DL}\rangle$ where KB_{LOG} is satisfiable *iff* KB_{DL} is satisfiable.

Remark 2 (Equivalence of configuration problems):

Let $CLANG_{LOG} = CLANG_{DL}$ where each concept is interpreted as monadic and each role is interpreted as dyadic predicate. $CONF_{LOG}$ describes the actual configuration by two sets of facts $CONF_{LOG} =$ *COMP-facts* $\cup$ *ROLE-facts*. The construction of $CONF_{LOG}$ is based on $CONF_{DL} = COMPS \cup ROLES$ where *COMP-facts* $= \{C_i(ci) | ci \in INDIVS_{C_i}, C_i \in C_{config}\}$ and *ROLE-facts* $= \{R_j(rj, sj) | \langle rj, sj\rangle \in TUPLES_{R_j}, R_j \in R_{config}\}$. $DD_{LOG} = \mathcal{T}\langle DD_{DL}\rangle$, and $SRS_{LOG} = \mathcal{T}\langle SRS_{DL}\rangle$.

$CONF_{LOG}$ is a valid configuration for $(DD_{LOG}, SRS_{LOG}, CLANG_{LOG})$ *iff* $CONF_{DL}$ is a valid configuration for $(DD_{DL}, SRS_{DL}, CLANG_{DL})$. □

Remark 2 follows from Remark 1 and the equivalence property of the translation function. Note that the completeness axioms correspond exactly to the translation of the axioms AX_{COMPS} and AX_{ROLES} by applying the translation function proposed in [5].

From the equivalence of configuration problems follow two important consequences. First, the two main streams in solving configuration problems based on description logics on the one hand and first order predicate or propositional logic on the other hand can be easily transformed to each other. Second, since description logics without transitive closure are equally expressive to dyadic predicate logic with at most three (counting) quantifiers [5] it follows that the predicate logic based approach is strictly more expressive than the description logics based approach, implying that certain logic constructions have to be simulated by more complex description logic constructions, and also that certain complex structural restrictions [6] will not be expressible in the language directly but will have to be incorporated using a more expressive assertional language.

4 Building configuration knowledge bases for the Semantic Web

In the following we show how to represent the configuration domain specific modeling concepts discussed in Section 2 using OIL.

Component types. The representation of component types is straightforward and has already been shown in Section 3. Component types are transformed into corresponding concepts, attributes into role definitions constraining datatype and cardinality (the cardinality of attribute roles is set to 1). Attributes as well as abstract roles are inherited by the defined subconcepts. In most configuration environments the semantics of a generalization hierarchy are disjunctive (*disjoint* concept) and complete by default (each instance of a supertype is also an instance of exactly one of its subtypes).

Part-whole relationships. Part-whole relationships play an important role in many application domains having quite different semantics (see [1], [16]). Basically, a part-whole relationship can be expressed using the two roles *PartOf* and *HasPart*, where *PartOf* is the inverse role of *HasPart*. In OIL or DAML+OIL we can define - depending on the application domain - different semantics for part-whole relationships by imposing restrictions on the usage of the corresponding roles. Sattler [16] presents an extension of the basic description logic $\mathcal{ALC}$ with concepts for adequate representation of part-whole relationships - in this context a categorization of different facets of part-whole relationships is given. In our working example (Figure 1) we only allow part-whole relationships where a component is part-of exactly one other component (this is also denoted as exclusive part-whole relationship). Such exclusivity restrictions can be introduced by restricting the cardinality of the corresponding role to 1, e.g., *slot-constraint mb-of-cpu cardinality 1 MB*, where the role *mb-of-cpu* must be interpreted as a subslot of the role *PartOf*. Furthermore, restrictions concerning the cardinality of parts must be added to the concept definition of the whole, e.g., *slot-constraint cpu-of-mb min-cardinality 1 CPU* (*max-cardinality 2 CPU*).

Port connections. As mentioned above, certain predicate logic constructs must be simulated by more complex description logic arrangements [5]. In terms of predicate logic, port connections can be represented using a predicate *conn*(C_1, P_1, C_2, P_2), i.e., component C_1 is connected via port P_1 with component C_2 via port P_2 - e.g., *conn*$(videocard1, videoport2, screen1, screenport1)$ describes a connection between $videoport2$ of $videocard1$ and $screenport1$ of $screen1$. In order to represent port-based connections, we have to introduce a *Port* concept, which is characterized by a role indicating the related component concept (role *compnt*) and a role which indicates the used portname of the connection (role *portname*) - additionally, a role *conn* indicates the connection to the second involved *Port* concept. Although a representation of port connections is possible with OIL or DAML+OIL, the original knowledge of domain experts is split into multiple pieces of knowledge which are hard for these domain experts to understand. Also, note that the connections via such a connection object must be unique and therefore the roles must be bidirectionally defined using the *inverse* role constructor. The constraint that a *Videocard* must be connected via $videoport2$ with a *Screen* via $screenport1$ can be formulated as

Example 6: *Videocard:*(*slot-constraint videoport has-value*((*slot-constraint portname has-value* (*one-of videoport2*)) *and* (*slot-constraint conn has-value* ((*slot-constraint compnt has-value Screen*) *and* (*slot-constraint portname has-value* (*one-of screenport1*)))))). □

The constituting elements of such port constraints are *navigations* representing role compositions between connected concepts.

Navigation in product structure. In the following we discuss a set of representative constraint concepts which are frequently used in the configuration domain [7, 17]. The constraints consist of navigation expressions over concepts which are represented by complex paths over abstract roles.

Definition 6 (Navigation expression): *Given two concepts* C_1 *and* C_2*, a navigation expression from* C_1 *to* C_2 *is formulated as a* *sequence of existential role quantifications*

(slot-constraint r_1 has-value(slot-constraint r_2 has-value ... (slot-constraint r_n has-value C_2))),

where <$r_1, r_2, ..., r_n$> denotes a sequence of roles along the navigation path from C_1 to C_2 (r_1 is a role of C_1 and C_2 is in the range of r_n). In the following we use the expression navpath(C_1, C_2) as short hand notation for a navigation path concept from C_1 to C_2. □

Definition 7 (Common root): *A concept C_R is denoted as common root of a set of concepts $C = \{C_1, ..., C_n\}$, if there exists a navigation path from C_R to each concept of C.* □

Incompatibilities. An incompatibility constraint between concepts C_1, C_2, ..., C_n can be expressed as C_R: *not*(*navpath*(C_R, C_1) *and navpath*(C_R, C_2) ... *and navpath* (C_R,C_n)), where C_R is the common root of C_1, C_2, ..., C_n in DD_{DL}.

Example 7 (IDEUnit incompatible with MB2): *Computer: not*((*slot-constraint hdunit-of-computer has-value IDEUnit*) *and* (*slot-constraint mb-of-computer has-value MB2*)) □

Requirements. A *requires* dependency between concepts C_1 and C_2 (C_1 requires C_2) can be expressed as C_R: *not*(*navpath*(C_R, C_1)) *or navpath*(C_R, C_2) with the common root C_R of C_1, C_2 in DD_{DL}. Similar to incompatibilities, requirements can be extended by introducing further navigation paths on the LHS and RHS of a requires dependency, e.g. $C_1 \wedge C_3$ *requires* $C_2 \vee C_4$.

Example 8 (SCSIUnit requires MB1): *Computer:*((*not*(*slot-constraint hdunit-of-computer has-value SCSIUnit*)) *or* (*slot-constraint mb-of-computer has-value MB1*)) □

Resource balancing. Resource constraints can be formulated using aggregation functions as proposed in [2]. We assume the existence of a value domain $dom(D)$, a set of predicate symbols P_i associated with binary relations (typically $<, \leq, =, \geq, >$) over $dom(D)$, and a set of aggregation functions $agg(D)$ (typically sum, avg, $count$, id; the last being *identity* in the case where we want to compare to a single fixed value instead of to an aggregate). Concerning the path leading to the concept whose feature values are aggregated, the definition of [2] requires that all but the last one of the roles in this path must be features.

Let <$r_1^c, r_2^c, ..., r_{n-1}^c, r_n^c$> and <$r_1^p, r_2^p, ..., r_{m-1}^p, r_m^p$> be navigation expressions with C_R as common root leading to the consumer (producer) concepts C_c (C_p). Furthermore, let f_c, f_p be features of C_c and C_p and $\Sigma \in agg(D)$ be an aggregation function. We can express a resource constraint as C_R: $P(r_1^c\, r_2^c\, ...\, r_{n-1}^c\, \Sigma(r_n^c \circ f_c), r_1^p\, r_2^p\, ...\, r_{m-1}^p\, \Sigma(r_m^p \circ f_p))$ according to [2].

Generally, if two aggregates are compared, one interprets the set that has to produce a smaller value as the set of *consumers*, and the set that has to produce a larger value as the set of *producers*.

Now we can define a concept *Computer* that comes equipped with a set of *HDUnits* that provide a corresponding hard-disk capacity and a set of software components that need hard-disk capacity. We express the requirement that the total hard-disk capacity consumed by software cannot exceed the installed hard-disk capacity. In this case navigation expressions only consist of two elements, consequently *hdcapacity* must be a feature, while *hdunit-of-computer* is a general role. The last line is the actual resource constraint.

Example 9 (Σ swcapacity $\leq$ Σ hdcapacity):
Software: slot-def swcapacity range (*min 0*) *properties functional*
HDUnit: slot-def hdcapacity range (*min 0*) *properties functional*
Computer: slot-def hdunit-of-computer range HDUnit
slot-def sw-of-computer range Software
lesseq (*sum*(*sw-of-computer*$\circ$*swcapacity*),
sum(*hdunit-of-computer*$\circ$*hdcapacity*)) □

Analysis. While the basic frame structure and formal basis of description logics based languages makes them one of the natural choice for configuration representation, certain demands on expressiveness must be met. The current versions of OIL and DAML+OIL do not support aggregation functions which are fundamental representation concepts frequently used in the configuration domain. Sattler and Baader [2] provided concrete domains and aggregation functions over them as extensions to the basic description logic $\mathcal{ALC}$. In addition to aggregation functions, built-in predicates must be allowed in order to support comparisons on the results of aggregation functions as well as on local features. Since trivial structural conditions lead to definitional overhead (e.g., when defining restrictions on port connections), additional concepts must be provided allowing the definition of roles with arity greater than two; the description of more complex structural properties (see [6]) would be supported by permitting the usage of variables. As far as the definition of rule languages for expressing detailed configuration knowledge on top of DAML+OIL is concerned, we must stress that such a language must allow disjuncts of positive literals when writing the constraint in clausal form, e.g., to express alternative port connections.

Happily, most of the required means of expression are already available in the Description Logic designer's toolbox; however, the degree of expressivity required also leads to problems w.r.t. to decidability of basic properties such as satisfiability or subsumption [2]. State-of-the-art configurators achieve decidability by putting a predefined limit on the number of individuals and allowing only finite domains of values for features (constraint variables). Furthermore, only fixed concept hierarchies are allowed (as part catalogues are typically considered unchangeable), which reduces the importance of subsumption versus that of A-Box reasoning.

5 Conclusions

In this paper we have shown how to apply Semantic Web ontology representation languages for configuration knowledge representation and integration. We gave a description logic based definition of a configuration problem and showed its equivalence with corresponding consistency-based definitions. A consequence of this equivalence is that configuration problems represented in description logics can easily be transformed into configuration problems represented in predicate logic or simplified variants thereof (and vice-versa). Consequently, we provide a common foundation that enables joint research activities and exploration of results. With respect to ongoing efforts to extend DAML+OIL our paper contributes a set of criteria which must be fulfilled in order to use such a language for full-fledged configuration knowledge representation. Thus DAML+OIL has the opportunity for being a standard knowledge representation language for the configuration domain.

References

[1] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-Whole Relations in Object-Centered Systems: An Overview. *Data & Knowledge Engineering*, 20(3):347–383, 1996.

[2] F. Baader and U. Sattler. Description Logics with Concrete Domains and Aggregation. In *Proceedings of the 13^{th} European Conference on Artificial Intelligence (ECAI '98)*, pages 336–340, Brighton, UK, 1998.

[3] 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.

[4] T. Berners-Lee. *Weaving the Web.* Harper Business, 2000.

[5] A. Borgida. On the relative expressive power of description logics and predicate calculus. *Artificial Intelligence*, 82:353–367, 1996.

[6] J. Cai, M. Furer, and N. Immerman. An optimal lower bound on the number of variables for graph identification. In *Proceedings of* 30^{th} *IEEE Symposium on FOCS*, pages 612–617, 1989.

[7] A. Felfernig, G. Friedrich, and D. Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 10(4):449–469, 2000.

[8] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-Based Diagnosis of Configuration Knowledge Bases. In *Proceedings of the* 14^{th} *European Conference on Artificial Intelligence (ECAI 2000)*, pages 146–150, Berlin, Germany, 2000.

[9] D. Fensel, F. vanHarmelen, I. Horrocks, D. McGuinness, and P.F. Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

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

[11] T. Gruber, R. Olsen, and J. Runkel. The configuration design ontologies and the VT elevator domain theory. *International Journal of Human-Computer Studies*, 44(3/4):569–598, 1996.

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

[13] D. Mailharro. A classification and constraint-based framework for configuration. *AI Engineering Design Analysis and Manufacturing Journal, Special Issue: Configuration Design*, 12(4):383–397, 1998.

[14] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 90)*, pages 25–32, Boston, MA, 1990.

[15] S. Mittal and F. Frayman. Towards a Generic Model of Configuration Tasks. In *Proceedings* 11^{th} *International Joint Conf. on Artificial Intelligence*, pages 1395–1401, Detroit, MI, 1989.

[16] U. Sattler. Description Logics for the Representation of Aggregated Objects. In *Proceedings of the* 14^{th} *European Conference on Artificial Intelligence (ECAI 2000)*, pages 239–243, Berlin, Germany, 2000.

[17] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a General Ontology of Configuration. *AI Engineering Design Analysis and Manufacturing Journal, Special Issue: Configuration Design*, 12(4):357–372, 1998.

[18] F. vanHarmelen, P.F. Patel-Schneider, and I. Horrocks. A Model-Theoretic Semantics for DAML+OIL. *www.daml.org*, March 2001.

[19] J.R. Wright, E. Weixelbaum, G.T. Vesonder, K.E. Brown, S.R. Palmer, J.I. Berman, and H.H. Moore. A Knowledge-Based Configurator that supports Sales, Engineering, and Manufacturing at AT&T Network Systems. *AI Magazine*, 14(3):69–80, 1993.

A Subsumption-Based Configuration Tool for Architectural Design

Dan Corbett[1]

Abstract. This paper discusses research in bringing formal structured knowledge representation to computational design in building architecture. Specifically, we demonstrate that the techniques of type subsumption, join and unification can be used as tools to turn a general design into a complete design. We have developed a software tool which uses these techniques to assist in the exploration of designs. The significance of this work is that we demonstrate that the merging of designs represented as Conceptual Graphs is efficient and useful to the designer.

1 SEARCH AND DESIGN

Designers have borrowed Artificial Intelligence techniques to explore design possibilities and models. While AI researchers may know these techniques as state space search, designers see them as discovery, or as guided movement through design possibilities [Woodbury et al. 2000] . The point of automated search for the designer is to use computer media that engage designers in exploring design modifications. The design user may want to create new designs, or index, compare or adapt existing designs. This type of user requires efficient representations for the designs and states (of designs) in a symbol system. The designer needs to be able to represent spaces of possibilities which are both relevant to the discovery process and lend themselves to tractable computations. It is necessary for the design process that the information in the system can be ordered by specificity, since design exploration usually means starting from an under-specified design and proceeding to a more specialized state.

This constraint has led us to consider state spaces structured by information specificity. Type hierarchies, subsumption and conceptual relations are used to realize these concepts. Using techniques and theory that we developed as part of a knowledge representation project, we developed software which will allow the design user to construct specific designs from generic designs (or new designs from old) while preserving all constraints on the process. We employ the techniques of Conceptual Graphs to represent and manipulate the design.

2 CONCEPTUAL GRAPHS

It has been demonstrated many times that graphs are a powerful and efficient knowledge representation technique. Mugnier and Chein [Mugnier and Chein 1996] illustrate quite effectively why labeled graphs are useful for knowledge representation in general. Among the main advantages that they list are a solid grounding when it comes to combinatorial algorithms, and that a graph (as a mathematical object) allows a natural representation (and therefore permits the construction of effective algorithms). There have been many attempts to formalize and standardize these graphical knowledge representation schemes, but probably none has been as extensive and comprehensive in recent times as Conceptual Structures.

Conceptual Structures (or Conceptual Graphs, or "CGs") are a knowledge representation scheme, inspired by the existential graphs of Charles Sanders Peirce and further extended and defined by John Sowa [Sowa 1984]. Informally, CGs can be thought of as a formalization and extension of Semantic Networks, although the origins are different. They are labeled graphs with two types of nodes: concepts (which represent objects, entities or ideas) and relation nodes, which represent relations between the concepts.

Every concept or relation has an associated type. A concept may also have a specific referent or individual. A concept in a CG may represent a specific instance of that type (eg, Felix is a specific instance, or individual, of type "cat") or we may choose only to specify the type of the concept. In the standard canonical formation rules for Conceptual Graphs, unbound concepts are existentially quantified. A relation may have zero or one incoming arcs, and any number of outgoing arcs. The type of the relation determines the number of arcs allowed on the relation. The arcs always connect a concept to a relation. Arcs cannot exist between concepts, or between relations.

A canon in the sense discussed here is the set of all CGs which are well-formed, and meaningful in their domain. Canonical formation rules specify how CGs can be legally built and guarantee that resulting CGs satisfy "sensibility constraints." The sensibility constraints are rules in the domain which specify how a CG can be built, for example that the concept *eats* must have a theme which is *food*. Note that canonicity does not guarantee validity. A CG may be well-formed in the canonical formation rules for the domain, but still be false.

A type hierarchy is established for both the concepts and the relations within a canon. A type hierarchy is based on the intuition that some types subsume other types, for example, every instance of "cat" would also have all the properties of "mammal." This hierarchy is expressed by a subsumption or generalization-order on types.

Formally, we define Conceptual Graphs as follows:

Definition 1. Conceptual Graph. A Conceptual Graph with respect to a canon is a tuple $G = (C, q, R, type, referent, arg_1, \ldots, arg_m)$ where

C is the set of concepts, $type : C \to T$ indicates the type of a concept, and $referent : C \to I$ indicates the referent marker of a

[1] Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Adelaide, South Australia 5095, email: corbett@cs.UniSA.edu.au

concept. The referent marker is either a pointer to an individual or a generic marker, which indicates that the individual is of the type indicated, but is existentially quantified.

T is the set of types. We will further assume that *T* contains two disjunctive subsets T_C and T_R containing types for concepts and relations.

I is the set of individuals.

q is a distinguished member of C, the head or root node of the graph.

R is the set of conceptual relations, $type : R \rightarrow T$ indicates the type of a relation, and each $arg_i : R \rightarrow C$ is a partial function where $arg_i(r)$ indicates the i-th argument of the relation *r*. The argument functions are partial as they are undefined for arguments higher than the relation's arity.

The set of types discussed in Definition 1 is arranged into a type hierarchy, ordered according to the specificity of each type. A type hierarchy is established for both the concepts and the relations within a canon. A type hierarchy is based on the intuition that some types subsume other types, for example, every instance of *cat* would also have all the properties of *mammal.* This hierarchy is expressed by a subsumption or generalization order on types. A type *t* is said to be more specific than a type *s* if *t* specializes some of the concepts from *s*.

An example of a relation type hierarchy is shown in Figure 1. In our domain of building architecture, we may wish to represent that one structure supports another structure. We may further want to represent that any type of support structure which supports a heavy load will also support a light load. This relationship is expressed in the hierarchy. In this manner, some constraints on the relations between concepts can be represented.

Similarly, an example type hierarchy for concepts is shown in Figure 2. The universal type is shown at the top of the hierarchy, and is represented by $\top$. The absurd type is shown at the bottom of the graph, and is represented by $\perp$. Here we see that a support structure is a specialization of a structure, and that a bay structure specializes support structure. Using these type hierarchies, it is possible to show, for example, that the multiple-bay structure will support a heavy load, by using concepts for multiple-bay structure, and a relation of the type supports-heavy.

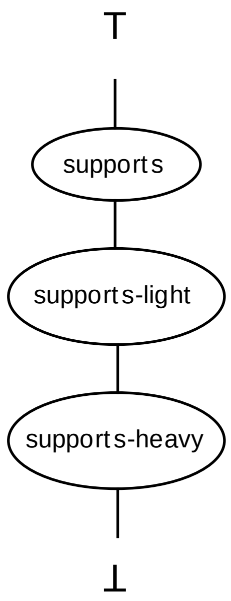


Figure 1. A relation type hierarchy.

Müller demonstrates in his work [Müller 1997] that it is not always possible to find a common specialization for two arbitrary graphs in the same canon, and therefore it is not possible to guarantee that two graphs can be unified. If a subset of CGs is defined which is all graphs that have a designated head node, and the head nodes are of compatible types, then unification of these graphs will at least be tractable. We define *q*, the head node of a graph, in order to be able to guarantee unification of the graphs under Müller's algorithms. This method allows us to combine the graphs while preserving the knowledge in both graphs, and still be able to use efficient unification methods as defined by Willems, Müller, and others [Willems 1995; Müller 1997].

The definitions of unification, consistency and type subsumption in this paper are based on formal concepts of projection and lower bounds. Carpenter [Carpenter 1992] defines each of these operators as a morphism. We have modified Carpenter's definitions to work with the properties of Conceptual Graphs. A morphism is then a mapping from the set of nodes of one Conceptual Graph to the set of nodes of another that preserves the order of relation arguments and the values of those arguments. In a morphism, all of the connections and arguments are preserved. The following definition of projection is the standard definition used in recent Conceptual Graph literature [Willems 1995; Mugnier and Chein 1996; Leclère 1997; Müller 1997; Corbett 2001].

Definition 2. Projection.
$G = (C, R, type, referent, arg_1, \ldots, arg_m)$ subsumes $G' = (C', R', type', referent', arg'_1, \ldots, arg'_m)$, $G \geq G'$, if and only if there is a pair of functions $h_C : C \rightarrow C'$ and $h_R : R \rightarrow R'$, called morphisms, such that:

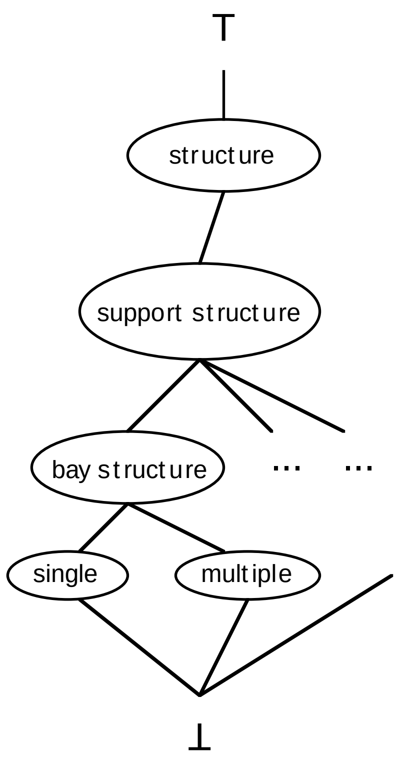


Figure 2. A concept type hierarchy.

$\forall c \in C$ and $\forall c' \in C'$, $h_C(c) = c'$ only if $type(c) \geq type'(c')$, and $referent(c) = *$ or $referent(c) = referent(c')$

$\forall r \in R$ and $\forall r' \in R'$ $h_R(r) = r'$ only if $type(r) \geq type'(r')$

$\forall r \in R$, $arg'_i(h_R(r)) = h_C(arg_i(r))$,

Willems also includes the following non-emptiness condition in his definition of projection in [Willems 1995]:

$\forall c \in C$ there is a concept $c' \in C'$, such that $h_C(c) = c'$

This non-emptiness condition guarantees that all the concepts present in the more general graph are also present in the more specific graph, although they may be in a more specific state. Willems' definition allows for the more specific graph to have concepts of a more specific type, or for a generic referent to be replaced by a specific individual. The definition used here also allows admits the non-emptiness condition.

Conceptual Graphs were chosen for this research because they can represent partial knowledge in a domain. It is possible for a designer to specify an object (or design) at some intermediate stage, with only very general types for the concepts and relations. When more specific knowledge becomes available, or when the designer chooses to fill in the details of a design, the operators of specialization, unification and join are employed.

In our work, *partialness* means that a structure need not contain all information that is implied about it by its structure and types. A partial representation is used here as a generalized, or higher-level description of an object in the domain. Whether a structure is partial or not depends on the context of the knowledge, and the domain. In domain terms, a model might be partial against one set of knowledge but complete with respect to a subset of the knowledge. For example, if our current domain knowledge of a building is limited to its spatial organization, a complete model of it would assign functions to physical spaces. Such a model would be partial with respect to a larger set of knowledge, containing for example, knowledge of how to construct the building.

The main thrust of the research described in this paper is the unification of Conceptual Graphs in terms of conjoining the knowledge contained in two different graphs. While this may involve term substitution (or the Conceptual Graphs equivalent - instantiation, subsumption, variable binding, etc.) and constraint solving, our work is more concerned with knowledge conjunction as discussed in [Carpenter 1992]. Carpenter defines unification as a system in which two pieces of partial information can be combined into a single unified whole. In our case, these pieces of partial information are represented by Conceptual Graphs. Carpenter refers to this idea as information conjunction, but in our work, it is *knowledge conjunction* that is more important. We want to be able to combine the expert knowledge of a system, not merely gather additional information. Unification here is the combining of pieces of knowledge in a domain, represented as Conceptual Graphs. We want to define unification as an operation that simultaneously determines the consistency of two pieces of partial or incomplete knowledge, and if they are consistent, combines them into a single result.

3 SPECIALIZATION OF DESIGNS

The standard techniques for creating more specialized concepts from general concepts are collectively known as the Canonical Formation Rules. The following definitions are standard, classical definitions of CG formation, which date back to Sowa's original 1984 work on Conceptual Graphs [Sowa 1984], but which were formalized much more recently [Wermelinger and Lopes 1994; Müller 1997]. We present here rules based on the work of Müller [Müller 1997].

1. **Join.** Given two CGs $G = (C, R, type, referent, arg_1, \ldots, arg_m)$ and $G' = (C', R', type', referent', arg'_1, \ldots, arg'_m)$ (without loss of generality we assume C and C' to be disjoint) $\forall c \in C$, and $\forall c' \in C'$ where c = c´ (that is, they have identical types and referents) or c ≥ c´ (ie, the type of c subsumes the type of c´) the external join of C and C' is the CG $G'' = (C \cup (C' - \{c'\}), R \cup (R'_{c':=c}), type'', referent'')$. The subscript $c':=c$ denotes the replacement of every occurrence of c´ by c.

2. **Restrict.** Given a CG $G = (C, R, type, referent, arg_1, \ldots, arg_m)$ and a node $c \in C$ with type t which has a subtype s ≠ ⊥the restrict type is the CG $G' = (C, R, type', referent)$ such that type´(c) = s, ∀d ≠ c: type´(d) = type(d).

Sowa [Sowa 1984] (and others) also define a *copy* rule, which allows a new graph G´ to be created as an exact duplicate of a graph G, and a *simplify* rule which allows the deletion of duplicate (and presumably redundant) relations.

We are able, then, to represent architectural designs as Conceptual Graphs. The architect can start with a general description of a design, and work through successive specializations to explore the design. Each specialization is subsumed by more general designs, and it is therefore guaranteed that each specialization conforms to all the constraints of the original, generic specification.

We now show that subsumption is just another form of the projection operator on Conceptual Graphs, defined previously.

Definition 3. Subsumption. We say that a Conceptual Graph G subsumes another Conceptual Graph G', or $G \geq_{CG} G'$, iff G' can be obtained by applying a finite number of canonical formation rules to G.

Since any application of the canonical formation rules to a graph s will always produce a graph t which is more specific than the original, s will necessarily have a projection into the new graph t. Chein and Mugnier formalize this idea, and demonstrate that s ≥ t iff there exists a projection from s to t [Chein and Mugnier 1992].

The unification of two conceptual graphs with constraints now becomes the combination of two graphs which are compatible in corresponding concepts and relations, as defined by our definition of the projection operator and join. Any constraints on the values in the concepts of the graph are preserved through the unification process by the projection operator.

Mugnier and Chein convincingly demonstrate that any Constraint Satisfaction problem (CSP) can be represented as a mathematical morphism [Mugnier and Chein 1996]. Their proof of the strong correspondence between CSP and the general problem of morphism (or projection) also demonstrates that a type hierarchy, such as used in Conceptual Graphs, can be effective in representing and solving a CSP problem. They develop, and prove the soundness of, algorithms for transferring CSP problems to a projection problem, and for transferring projections back to a CSP representation.

Mugnier and Chein demonstrate that the algorithmic techniques that they develop for resolving the problem of the existence of a solution to a Constraint Satisfaction Problem also can enumerate the solutions. Further, these are transferable from one domain to another [Mugnier and Chein 1996].

Formal definitions and algorithms for unification are discussed in detail in [Willems 1995; Corbett and Woodbury 1999; Corbett 2001]. A complete discussion and formal treatment of our unification algorithm is presented in [Corbett 2001]. Rather than repeat these here, we proceed to the discussion of the use of the configuration tool in the Architecture domain.

4 DESIGN EXAMPLES

For our example domain, we detail the SEED project. The intent of the SEED project is to create software which will support preliminary design of buildings [Heisserman 1991; Heisserman 1995; Chang and Woodbury 1996; Corbett and Burrow 1996; Woodbury et al. 1999; Woodbury et al. 2000] . This includes using the computer as an active tool which helps to generate designs [Flemming and Woodbury 1995] . The SEED project architects were willing to test the software that was produced from our work.

SEED is an acronym for **S**oftware **E**nvironment to support the **E**arly phases in building **D**esign. Specifically, SEED will help with recurring building types (designs which are used again frequently). The SEED system is intended to be an aid to architects in creating building designs by reusing design knowledge. In order to store and reuse the design cases in an efficient manner, it is necessary to use a representation scheme which can handle real-value constraints and unification.

The SEED system is built around the idea of a design space. The design space is a set of partial or complete solutions to an architectural design problem. In this sense, it is roughly equivalent to the AI term "search space", in that the design space is defined by starting states and operators which allow the derivation of one state from another, including some acceptable goal states.

SEED works by exploring the design space during the elaboration of a design. To achieve the goal of design experience reuse, SEED allows for the storage of “interesting” design states, where “interesting” is decided either by the user, or by the interaction of the user's search path and heuristics in SEED. The difficulty, then, is in identifying and retrieving stored design states containing design decisions most applicable to the current state, that is, retrieving useful information corresponding to design experience captured in the historical pattern of explored and stored design states.

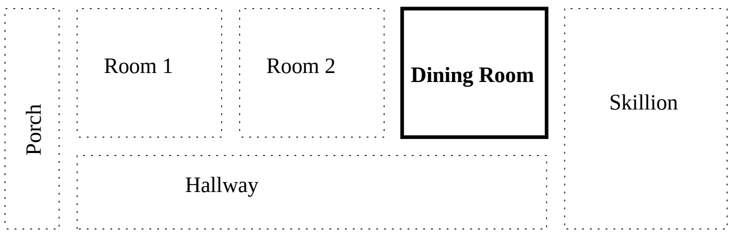


Figure 3. Basic floor plan of a single-fronted cottage.

As SEED explores the design space, each of the retrieved designs must be compared to the requirements to find whether the design meets each of the specifications and constraints. The problem then, is to find a previous design which will unify with the specification currently being worked on. This unification process must attempt to identify each attribute of the specification with the same attribute in the retrieved design. If all the attributes can be unified while satisfying all of the constraints, then the two structures are said to unify, producing a new structure which is a combination of the knowledge in the two previous structures. The new structure is a new design, which can be used to satisfy the current design requirements.

Appealing to our example domain, Figure 3 shows the basic floor plan of a single-fronted cottage, a design which is very common in many parts of Australia. Figure 4 shows the corresponding Conceptual Graph. The graph represents the general structure of the cottage (and some adjacency relations) but with none of the details that will make this design into a realized cottage. (The ellipses indicate further parts of the graph which we do not have space to elaborate here.) A designer may retrieve a previous design from a library of designs to provide some of the details. An example of such a previous design is shown in Figure 5. Here, a design for a dining room adjacent to a living room is selected as a partial match for the general design. Many of the details of the rooms have already been specified. This partial design can be unified with the graph in Figure 4, using the methods described in this paper. There are four possible locations for attaching the partial design to the general design. Our intent is that the system will not make the choice of which nodes to unify, leaving the choice to the designer.

Since SEED works on the principle of constructive design, it is important to be able to create small units in the design, and then link them together. The mechanism we use to link these units is unification. For example, the partial design illustrated in Figure 6 contains the concept *kitchen*. This may initially be used as a single concept, or as a link to the standard or template attributes of a kitchen. These generic concepts would be specialized later. Figure 7 shows a design for a kitchen, with some of the usual relations.

The *area* relation indicates that the value of the area is an

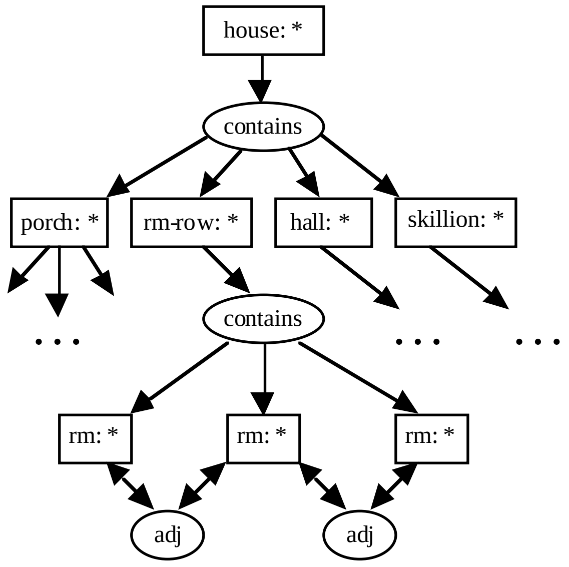


Figure 4. Conceptual Graph representation of Figure 3.

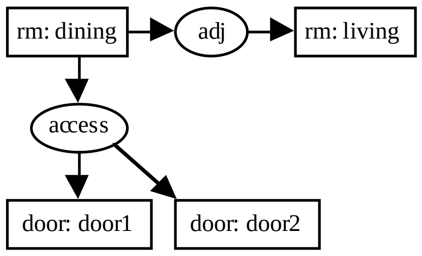


Figure 5. A specialization of two rooms.

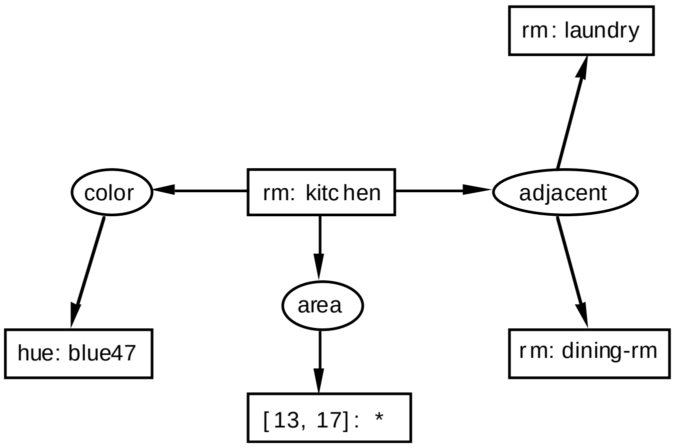


Figure 6. A Conceptual Graph representation of a kitchen.

interval, which constrains the values that the area may have. Other attributes may include insulation, illumination, support structure and other factors which concern the design of a building structure. We assume that these types and their respective type hierarchies have already been defined, with their obvious meanings.

The unification of two design Conceptual Graphs is another graph representing neither more nor less information than is contained in the two graphs being unified. Unification only fails when it is applied to designs that, when taken together, provide inconsistent information. In the case of the design domain, inconsistency can only arise from attempting to unify two structures that assign incompatible values (in either literal information, or in the lattices defined for that type of information) to the same attribute.

Figure 7 is an example of a kitchen SU specified by the user to have one adjacent room and a floor area constrained to be between fifteen and twenty square meters, represented by the interval [15, 20]. When we try to find a match for the specified partial design among the previous designs, we retrieve the Conceptual Graph shown in Figure 6 as a previously-designed kitchen from the knowledge base. This is unified with the graph shown in Figure 7, and the result is as shown in Figure 8.

The *adjacent* relations unify by taking on the values of both "dining-rm" and "laundry", since these two values are compatible (ie there is nothing to exclude the kitchen from being adjacent to both the laundry and the dining room). The area relation unifies, because the intervals specified in the two original graphs have a join on the interval lattice. The join becomes the value of the unified area relation. The color relation unifies trivially, as there is

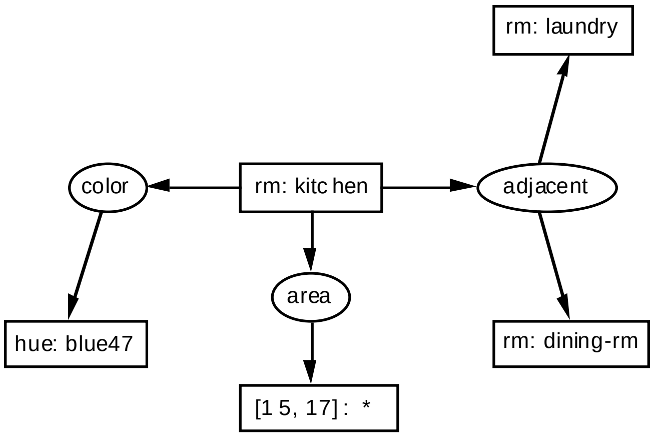


Figure 8. The unified design.

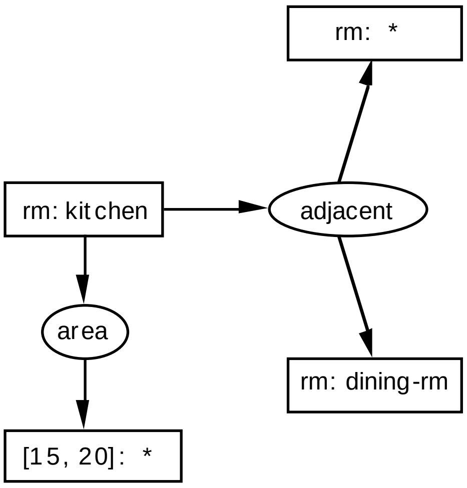


Figure 7. Another kitchen Conceptual Graph.

nothing specified which could be incompatible with it.

5 DISCUSSION

In discussions with the architects who helped to test the system, several issues of unification, constraints and matching were identified. The three main areas where the architectural designer needs the contribution of Conceptual Graph unification are in type subsumption, knowledge-level reasoning, and pattern matching. Each of these three areas is discussed below, along with a qualitative judgment of how well Conceptual Graphs and unification deal with these concerns.

The architects want to be able to use type subsumption to make statements such as, “An office (or kitchen, or corridor) is a kind of room. All the properties which apply to one should apply to its specializations.” This is distinct from the object-oriented objective of objects inheriting all the properties of a class of objects. The essential difference is in treating a kitchen as you would any generic room. A generic room can be placed, occupy space, and have attributes such as insulation and number of doors. A *class* of rooms will have attributes, but cannot be said to occupy a space or have specific dimensions, or have a specific count or placement of doors. The generic room can have constraints placed on its attributes, and finally can be specialized into a kitchen.

Fundamentally, a generic room can take the place of a specialized room, unlike a class of objects. The room can stay generic for as long as the user needs it to be generic, and then specialized. Further, the room could be specialized wholly or in part. If partly specified, it can be matched against other specifications to find appropriate matches.

Conceptual Graphs and the unification algorithm give this ability to the architects. The software allows the user to specialize designs by matching (unifying) previous designs with the current design problem. Since all characteristics, attributes and constraints are carried along in the unification, the specialization represents all of the design concepts included in the more generic design. Further, and more importantly, there is no real separation between generic and specific, since all points in between can be represented. Conceptual Graphs combined with the ability to specialize using unification are the ideal tool for the knowledge combination approach and the constructive nature of architectural design.

The second major concern of the architectural designers was the ability to have knowledge-level reasoning. That is, they want to be able to speak in the language of the architect, not the language of the computer (or CAD system). The user wants to be able to refer to the “north wall” or “door” without resorting to discussing geometric coordinates in space. The user wants to depart from previous CAD-based data-level processing, and work at the knowledge level in the architecture domain.

This is certainly another area where Conceptual Graphs and unification combine to bring a solution to this domain. While spatial coordinates (and their constraints) can be stored in a graphical representation of a room, there is no need for the user to bother with using them. The graph can be manipulated as a whole, and treated as a room, rather than a square in a diagram. The software system does not deal with lines and boxes, but rather with specializing entire designs for rooms (or houses, or office buildings). This approach frees the architect from dealing with data-level concerns of numbers and coordinates, and allows the architect instead to deal with the architectural design.

The third major concern of the architectural testing team is in the area of pattern matching. The users want to be able to start with a high-level, generic description of a building, perhaps represented as a hierarchy of design units. Then they want to be able to make queries such as, “Can this bay structure be used in the support structure?” or, “Do the constraints match up adequately for a particular technology to be used? If yes, tell me the constraints under which it is usable.”

Once again, the work presented here meets the requirements of the architects. A query can be represented as a Conceptual Graph. The user can specify a type of structure for support, and make the query by attempting to unify the structure with the more generic design. If the unification fails, then the user knows that the proposed structure does not meet the constraints of the design problem. If the graphs unify, then the resulting graph will contain the constraints which must be met in order to make the design work.

Overall, the system of unification over constraints on Conceptual Graphs presented here gives a set of tools to the designer. The ability to use knowledge combination with constraints to handle objects at the knowledge level greatly leverages the ability of the designer to work efficiently.

The software which implements this system is far from being complete, but the results are consistently as good as those presented in this paper. Several constraints on real values, structure and type can be resolved simultaneously in a system. The software will also allow variables in the place of constraints (such as [10, n]) and will allow a relaxation of constraints (for example, if the interval is only out by 10%). We have not examined the issue of the compounding of problems caused by relaxing multiple constraints simultaneously.

The software currently has a rough user interface, accepting and displaying Lisp lists. However, note our work with a new CG language which has a very friendly user interface in [Benn and Corbett 2001]. The pCG system, used as a front-end to the tools described in this paper, will provide an easy user access.

6 CONCLUSIONS

We have demonstrated a method for the formal representation of general architectural designs. The use of Conceptual Graphs is an efficient method for representing not only the designs, but also constraints on the designs and knowledge conjunction of designs. Type hierarchies and the canonical formation rules efficiently specialize the graphs into concrete designs.

The system described in this paper allows general designs to be represented as concepts, and also allows those values to be constrained by specifying valid intervals and type hierarchies. In our software, we also use inequality relations and allow variables in the constraint specifications. The experiments with this software have shown these techniques to be useful and efficient.

The significance of our work is that a simple unification operation, using join and type subsumption (implemented as projection on Conceptual Graphs), is defined which can be used to validate the constraints over an entire unified graph. Such a graph can be used to efficiently represent building designs.

7 ACKNOWLEDGEMENT

The author expresses his sincere gratitude to the architects in the SEED project who participated in discussions on, and assisted in the testing of the Conceptual Graph unification software: Rob Woodbury, Andrew Burrow, Sambit Datta and Teng-Wen Chang.

REFERENCES

Benn, D. and D. Corbett (2001). "An Implementation of the Process Mechanism and an Extensible CG Programming Language". In *Proc. Workshop on Conceptual Graph Tools, International Conference on Conceptual Structures*, Palo Alto, California, USA, August, 2001.

Carpenter, B. (1992). *The Logic of Typed Feature Structures*. Cambridge, Cambridge University Press, 1992.

Chang, T.-W. and R. F. Woodbury (1996). "Sufficiency of the SEED Knowledge-Level Representation for Grammatical Design". In *Proc. Australian New Zealand Conference on Intelligent Information Systems*, Adelaide, Australia, IEEE Press, November, 1996.

Chein, M. and M.-L. Mugnier (1992). "Conceptual Graphs: Fundamental Notions." *Revue d'Intelligence Artificielle* **6**(4): 365-406.

Corbett, D. R. (2001). "Conceptual Graphs with Constrained Reasoning." *Revue d'Intelligence Artificielle* **15**(1): 87-116.

Corbett, D. R. and A. L. Burrow (1996). "Knowledge Reuse in SEED Exploiting Conceptual Graphs". In *Proc. Supplemental Proceedings of the Fourth International Conference on Conceptual Structures*, Sydney, NSW, Australia, UNSW Press, August, 1996.

Corbett, D. R. and R. F. Woodbury (1999). "Unification over Constraints in Conceptual Graphs". In *Proc. Seventh International Conference on Conceptual Structures*, Blacksburg, Virginia, USA, Springer-Verlag, July, 1999.

Flemming, U. and R. F. Woodbury (1995). "Software Environment to Support Early Phases in Building Design (SEED) Overview." *Architectural Engineering* **1**(1).

Heisserman, J. A. (1991). *Generative Geometric Design and Boundary Solid Grammars*. PhD Thesis, Department of Architecture, Carnegie Mellon University, Pittsburgh, Penn, USA, 1991.

Heisserman, J. A. (1995). "Generative Geometric Design." *IEEE Computer Graphics and Applications* **14**(2): 37-45.

Leclère, M. (1997). "Reasoning with Type Definitions". In *Proc. Fifth International Conference on Conceptual Structures*, Seattle, Washington, USA, Springer-Verlag, August, 1997.

Mugnier, M.-L. and M. Chein (1996). "Représenter des Connaissances et Raisonner avec des Graphes." *Revue d'Intelligence Artificielle* **10**(6): 7-56.

Müller, T. (1997). *Conceptual Graphs as Terms: Prospects for Resolution Theorem Proving*. Masters Thesis, Department of Computer Science, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, 1997.

Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Reading, Mass, Addison-Wesley, 1984.

Wermelinger, M. and J. G. Lopes (1994). "Basic Conceptual Structures Theory". In *Proc. Second International Conference on Conceptual Structures*, Maryland, Springer-Verlag, August, 1994.

Willems, M. (1995). "Projection and Unification for Conceptual Graphs". In *Proc. Third International Conference on Conceptual Structures*, Santa Cruz, California, USA, Springer-Verlag, August, 1995.

Woodbury, R., S. Datta and A. L. Burrow (2000). "Erasure in Design Space Exploration". In *Proc. Artificial Intelligence in Design*, Worcester, Massachusetts, USA, June, 2000.

Woodbury, R. F., A. L. Burrow, S. Datta and T. W. Chang (1999). "Typed Feature Structures in Design Space Exploration." *Journal of Artificial Intelligence in Engineering, Design and Manufacturing* **13**(4): 287-302.

TCP-Nets for Preference-based Product Configuration

Ronen I. Brafman and **Carmel Domshlak**[1]

Abstract. A good configuration not only satisfies all imposed constraints, but does so in the manner most desirable by the user. Thus, to make good configuration choices, we must have some information about the potential user's preferences on alternative designs. In many applications, preference elicitation is a serious bottleneck. The user either does not have the time, the knowledge, or the expert support required to specify complex multi-attribute utility functions. In such cases, a method that is based on intuitive, yet expressive, preference statements is required. In this paper we suggest the use of TCP-nets, an enhancement of CP-nets, as a tool for representing, structuring, and reasoning about qualitative preference statements. We present and motivate this framework, define its semantics, and show why it is particularly suitable for the task of configuration.

1 INTRODUCTION

The ability to make decisions and to assess potential courses of action is a corner-stone of many AI applications in general, and of configuration problems in particular. To make good decisions, we must be able to assess and compare different alternatives. Sometimes, this comparison is performed implicitly, as in many recommender systems. However, in many cases explicit information about the decision-maker's preferences is required.

In classical decision theory and decision analysis a utility function is used to represent the decision-maker's preferences. Utility functions are a powerful form of knowledge representation. They provide a quantitative measure of the desirability of different outcomes, capture attitude toward risk, and support decision making under uncertainty. However, the process of obtaining the type of information required to generate a good utility function is involved and time-consuming and requires non-negligible effort on the part of the user. In some application, this effort is necessary and/or possible, e.g., when either uncertainty plays a key role, or the stakes involved are high, and when the decision-maker and the decision analyst are able and willing to engage in the required preference elicitation process. One would expect to see such effort invested when, for example, medical decisions are involved. However, there are many applications where either uncertainty is not a crucial factor, or the user cannot be engaged for a lengthy period of time (e.g., in on-line product recommendation systems), or the preference elicitation process cannot be supported by a human decision analyst and must be performed by a software system (e.g., because of replicability or mass marketing aims). In such cases, elicitation of a good utility function is not a realistic option.

When a utility function cannot be or need not be obtained, one should resort to other, more qualitative forms of preference representation. Ideally, this qualitative information should be easily obtainable from the user by non-intrusive means. That is, we should be able to generate it from natural and relatively simple statements about preferences obtained from the user, and this elicitation process should be amenable to automation. In addition, automated reasoning with this representation should be feasible and efficient.

One relatively recent framework for preference representation that addresses these concerns is that of *Conditional Preference Networks* (CP-nets) [4]. In CP-nets, the decision maker is asked to describe how her preference over the values of one variable depends on the value of other variables. For example, she may state that her preference for a dessert depends on the value of the main-course as well as whether or not she had an alcoholic beverage. Her choice of an alcoholic beverage depends on the main course and the time of day. This information is described by a graphical structure in which the nodes represent variables of interest and the edges represent dependence relations between the variables. Each node is annotated with a *conditional preference table* (CPT) describing the user's preference over alternative values of this node given different values of the parent nodes.

CP-nets capture a class of intuitive and useful natural language statements of the form "I prefer the value x_0 for variable X given that $Y = y_0$ and $Z = z_0$". Such statements do not require complex introspection nor a quantitative assessment. However, there is another class of statements that is no less intuitive or important. They have the following form: "It is more important to me that the value of X be high than that the value of Y be high." We call these *relative importance* statements. For instance, one might say "The length of the journey is more important to me than the choice of airline". A more refined notion, though still intuitive and easy to communicate, is that of *conditional relative importance*: "The length of the journey is more important to me than the choice of airline provided that I am lecturing the following day. Otherwise, the choice of airline is more important." This latter statement is of the form: "A better assignment for X is more important than a better assignment for Y given that $Z = z_0$." Notice that information about relative importance is different from information about independence. In the example above, my preference for an airline does not depend on the duration of the journey because, e.g., I compare airlines based on their service and security levels and the quality of their frequent flyer program.

In this paper we present an extension of CP-nets, which we call TCP-nets (for *tradeoffs-enhanced* CP-nets), show how they can be used to compute optimal outcomes given constraints, and discuss their applicability to the process of product configuration. TCP-nets capture both information about conditional independence and about conditional relative importance. Thus, they provide a richer framework for representing user preferences, allowing stronger conclusions to be drawn, yet remain committed to the use of intuitive, qualitative information as their source. Naturally, one can consider relative importance assessments among more than two variables. However,

[1] Both authors are from Computer Science Dept., Ben-Gurion University,Beer-Sheva, Israel, email: {brafman,dcarmel}@cs.bgu.ac.il

we feel that such statements are somewhat artificial and less natural to articulate.

This paper is organized as follows. Section 2 describes the notions underlying TCP-nets: preference relations, preferential independence, and relative importance. In Section 3 we define TCP-nets, and provide a number of examples. Section 4 shows how TCP-nets can be used to perform constrained optimization. Section 5 provides a discussion of using TCP-nets in product configuration. In this short version, we emphasize intuitions and motivation. The technically full version of this paper [5] contains the formal semantics of TCP-nets, discussion of consistency analysis for various TCP-nets, and a fuller discussion of the optimization problem.

2 PREFERENCE ORDERS, INDEPENDENCE, AND RELATIVE IMPORTANCE

In this section we describe the ideas underlying TCP-nets: preference orders, preferential independence and conditional preferential independence, as well as relative importance and conditional relative importance.

2.1 Preferences and Independence

A *preference relation* is a total pre-order (a *ranking*) over some set of outcomes. Given two outcomes o, o', we write $o \succeq o'$ to denote that o is at least as preferred as o' and we write $o \succ o'$ to denote that o is strictly more preferred than o'. Finally, if o and o' are equally preferred, we write $o \equiv o'$.

The types of outcomes we are concerned with consist of possible assignments to some set of variables. More formally, we assume some given set $\mathbf{V} = \{X_1, \ldots, X_n\}$ of variables with corresponding domains $\mathcal{D}(X_1), \ldots, \mathcal{D}(X_n)$. The set of possible outcomes is then $\mathcal{D}(X_1) \times \cdots \times \mathcal{D}(X_n)$. For example, in the context of the problem of configuring a personal computer (PC), the variables may be *processor type, screen size, operating system* etc., where *screen size* has the domain {*17in, 19in, 21in*}, *operating system* has the domain {*LINUX, Windows98, WindowsXP*}, etc. Each assignment to the set of variables specifies an outcome – a particular PC configuration. Thus, an ordering over these outcomes specifies a ranking over possible PC configurations.

The number of possible outcomes is exponential in n, while the set of possible total orders on them is doubly exponential in n. Therefore, an explicit representation and an explicit specification of a ranking are not realistic. We must find implicit means of describing this preference relation. Often, the notion of preferential independence plays a key role in such representations. Intuitively, $\mathbf{X}$ and $\mathbf{Y} = \mathbf{V} - \mathbf{X}$ are *preferentially independent* if for all assignments to $\mathbf{Y}$, our preference over $\mathbf{X}$ values are identical. More formally, let $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{D}(\mathbf{X})$ for some $\mathbf{X} \subseteq \mathbf{V}$ (where we use $\mathcal{D}(\cdot)$ to denote the domain of a set of variables as well), and let $\mathbf{y}_1, \mathbf{y}_2 \in \mathcal{D}(\mathbf{Y})$, where $\mathbf{Y} = \mathbf{V} - \mathbf{X}$. We say that $\mathbf{X}$ is *preferentially independent* of $\mathbf{Y}$ iff, for all $\mathbf{x}_1, \mathbf{x}_2, \mathbf{y}_1, \mathbf{y}_2$ we have that

$$\mathbf{x}_1\mathbf{y}_1 \succeq \mathbf{x}_2\mathbf{y}_1 \text{ iff } \mathbf{x}_1\mathbf{y}_2 \succeq \mathbf{x}_2\mathbf{y}_2$$

For example, in our PC configuration example, the user may assess *screen size* to be preferentially independent of *processor type* and *operating system*. This could be the case if, for instance, the user always prefers a larger screen to a smaller screen, no matter what the processor or the OS are.

Preferential independence is a strong property, and therefore, less common. A more refined notion is that of conditional preferential independence. Intuitively, $\mathbf{X}$ and $\mathbf{Y}$ are *conditionally preferentially independent* given $\mathbf{Z}$ if for every fixed assignment to $\mathbf{Z}$, the ranking of $\mathbf{X}$ elements is independent of the value of the $\mathbf{Y}$ elements. Formally, let $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ be a partition of $\mathbf{V}$ and let $\mathbf{z} \in \mathcal{D}(\mathbf{Z})$. $\mathbf{X}$ and $\mathbf{Y}$ are *conditionally preferentially independent* given $\mathbf{z}$ iff, for all $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{y}_1$, $\mathbf{y}_2$ we have that

$$\mathbf{x}_1\mathbf{y}_1\mathbf{z} \succeq \mathbf{x}_2\mathbf{y}_1\mathbf{z} \text{ iff } \mathbf{x}_1\mathbf{y}_2\mathbf{z} \succeq \mathbf{x}_2\mathbf{y}_2\mathbf{z}$$

$\mathbf{X}$ and $\mathbf{Y}$ are conditionally preferentially independent given $\mathbf{Z}$ if they are conditionally preferentially independent given any assignment $\mathbf{z} \in \mathcal{D}(\mathbf{Z})$. Returning to our PC example, the user may assess *operating system* to be independent of all other features given *processor type*. That is, it always prefers LINUX given an AMD processor and Windows98 given an Intel processor (e.g., because he might believe that Windows98 is optimized for the Intel processor, whereas LINUX is otherwise better).

2.2 Relative Importance

Although statements of preferential independence are natural and useful, the orderings obtained by relying on them alone are relatively weak. To understand this, consider two preferentially independent boolean attributes A and B with values a_1, a_2 and b_1, b_2, respectively. If A and B are preferentially independent, then we can specify a preference order over A values, say $a_1 \succ a_2$, independently of the value of B. Similarly, our preference over B values, say $b_1 \succ b_2$, is independent of the value of A. From this we can deduce that a_1b_1 is the most preferred outcome and a_2b_2 is the least preferred outcome. However, we do not know the relative order of a_1b_2 and a_2b_1. This is typically the case when we consider independent variables. We can rank each one given a fixed value of the other, but often, we cannot compare outcomes in which both values are different. One type of information that can address some (though not necessarily all) such comparisons is information about relative importance. For instance, if we say that A is more important than B then this means that we prefer to reduce the value of B rather than reduce the value of A. In that case, we know that $a_1b_2 \succ a_2b_1$, and we can (totally) order the set of outcomes as follows

$$a_1b_1 \succ a_1b_2 \succ a_2b_1 \succ a_2b_2.$$

Returning to our PC configuration example, suppose that *operating system* and *processor type* are independent attributes. We might say that *processor type* is more important than *operating system*, e.g, because we believe that the effect of the processor's type on system performance is much more important than the effect of the operating system.

Formally, let X and Y be preferentially independent given $\mathbf{W} = \mathbf{V} - \{X, Y\}$. We say that X is *more important* than Y, denoted by $X \rhd Y$, if for every assignment $\mathbf{w} \in \mathcal{D}(\mathbf{W})$ and for every x_i, x_j, y_a, y_b such that $x_i \succ x_j$ given $\mathbf{w}$ and $y_b \succ y_a$ given $\mathbf{w}$, we have that:

$$x_iy_a\mathbf{w} \succ x_jy_b\mathbf{w}.$$

Notice that this is a strict notion of importance – any reduction in Y is preferred to any reduction in X.[2] When both X and Y are binary variables and $x_1 \succ x_2$ and $y_1 \succ y_2$ hold given $\mathbf{w}$ then $X \rhd Y$ iff we have $x_1y_2\mathbf{w} \succ x_2y_1\mathbf{w}$ for all $\mathbf{w} \in \mathcal{D}(\mathbf{W})$.

[2] We note that this idea can be refined by providing an actual ordering over elements of $\mathcal{D}(XY)$. We have decided not to pursue this option farther because it is less natural to specify. However, our results generalize to such specifications as well.

Relative importance information is a natural enhancement of independence information. It retains the property we value so much: it corresponds to statements that a naive user would find simple and clear to evaluate and articulate. Moreover, it can be generalized naturally to a notion of conditional relative importance. For instance, suppose that the relative importance of *processor type* and *operating system* depends on the primary usage of the PC. For example, when the PC is used primarily for graphical applications, then the choice of an operating system is more important than that of a processor because certain important software packages for graphic design are not available on LINUX. However, for other applications, the processor type is more important because applications for both Windows and LINUX exist. Thus, we say that X is more important than Y given $\mathbf{z}$ if we always prefer to reduce the value of Y rather than the value of X when $\mathbf{z}$ holds.

Formally, let $X, Y, \mathbf{W}$ be as above, and let $\mathbf{Z} \subseteq \mathbf{W}$. We say that X is *more important* then Y given an assignment $\mathbf{z} \in \mathcal{D}(\mathbf{Z})$ (*ceteris paribus*) iff, for any assignment $\mathbf{w} \in \mathbf{W} = \mathbf{V} - (\{X, Y\} \cup \mathbf{Z})$ we have:

$$x_i y_a \mathbf{zw} \succ x_j y_b \mathbf{zw}$$

whenever $x_i \succ x_j$ given $\mathbf{zw}$ and $y_b \succ y_a$ given $\mathbf{zw}$. We denote this relation by $X \rhd_{\mathbf{z}} Y$.

Finally, if for some $\mathbf{z} \in \mathcal{D}(\mathbf{Z})$ we have that either $X \rhd_{\mathbf{z}} Y$, or $Y \rhd_{\mathbf{z}} X$, then we say that the relative importance of X and Y is conditioned on $\mathbf{Z}$, and write $\mathcal{RI}(X, Y, \mathbf{Z})$.

3 TCP-NETS

TCP-nets (for *conditional preference networks with tradeoffs*) are a graph-based representation that encodes statements of (conditional) preferential independence and (conditional) relative importance. We use this graph-based representation for two reasons: First, it is an intuitive visual representation of preference independence and relative importance statements. Second, the structure of the graph has important consequences to issues such as consistency and complexity of reasoning. For instance, one of the basic results we present in [5] shows that when this structure is "acyclic," (for a suitable definition of this notion) then the preference statements contained in the graph are consistent – that is, there is a total pre-order that satisfies them.

TCP-nets are annotated graphs with three types of edges. The nodes of a TCP-net $\mathcal{N}$ correspond to the problem variables $\mathbf{V}$. The first type of (directed) edges signifies preferential dependence. The existence of such an edge from X to Y implies that the user has different preferences over values of X given different values of Y. The second type of (directed) edges captures relative importance relations. The existence of such an edge from X to Y implies that X is more important than Y. Finally, the third, undirected, edge type signifies conditional importance relations. Nodes X and Y are linked if there exists some $\mathbf{Z}$ for which $\mathcal{RI}(X, Y, \mathbf{Z})$ holds.

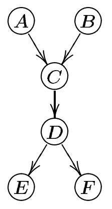
Each node X in a TCP-net is annotated with a *conditional preference table* (CPT). This table associates a preferences over $\mathcal{D}(X)$ for every possible value assignment to the parents of X (denoted $Pa(X)$). Each undirected arc is annotated with a *conditional importance table* (CIT). The CIT associated with the arc between X and Y describe the relative importance of X and Y given the possible value of the conditioning variables.

Formally, a TCP-net $\mathcal{N}$ is a tuple $\langle \mathbf{V}, \mathsf{cp}, \mathsf{i}, \mathsf{ci}, \mathsf{cpt}, \mathsf{cit} \rangle$, where

1. $\mathbf{V}$ is a set of nodes, corresponding to the problem variables.
2. cp is a set of directed cp-*arcs* $\{\alpha_1, \ldots, \alpha_k\}$ (where cp stands for *conditional preference*). A cp-arc $\langle \overrightarrow{X_i, X_j} \rangle$ belongs to $\mathcal{N}$ iff the preferences over the values of X_j depend on the actual value of X_i.
3. i is a set of directed i-*arcs* $\{\beta_1, \ldots, \beta_l\}$ (where i stands for *importance*). An i-arc $(\overrightarrow{X_i, X_j})$ belongs to $\mathcal{N}$ iff $X_i \rhd X_j$.
4. ci is a set of undirected ci-*arcs* $\{\gamma_1, \ldots, \gamma_m\}$ (where ci stands for *conditional importance*). A ci-arc (X_i, X_j) belongs to $\mathcal{N}$ iff we have $\mathcal{RI}(X_i, X_j, \mathbf{Z})$ for some $\mathbf{Z} \subseteq \mathbf{V} - \{X_i, X_j\}$.
5. cpt associates a CPT with every node $X \in \mathbf{V}$. A CPT is from $\mathcal{D}(Pa(X))$ (i.e., assignment's to X's parent nodes) to total pre-orders over $\mathcal{D}(X)$.
6. cit associates with every ci edge (X_i, X_j) a subset $\mathbf{Z}$ of $\mathbf{V} - \{X_i, X_j\}$ and a mapping from a subset of $\mathcal{D}(\mathbf{Z})$ to total orders over the set $\{X_i, X_j\}$. We call $\mathbf{Z}$ the *selector set* of (X_i, X_j) and denote it by $\mathcal{S}(X_i, X_j)$.[3]

We note that the following holds for every node X in the graph: X is independent of all other nodes given $Pa(X)$.

In the rest of this section we provide examples of TCP-net. We start with an example of a CP-net shown in Figure 1. A CP-net is a TCP-net in which the sets i and ci (and therefore cit) are empty.



$[a_1 \succ a_2]$
$[b_2 \succ b_1]$
$[(a_1 \wedge b_1) \vee (a_2 \wedge b_2) : c_1 \succ c_2;$
$\ (a_1 \wedge b_2) \vee (a_2 \wedge b_1) : c_2 \succ c_1]$
$[c_1 : d_1 \succ d_2;\ c_2 : d_2 \succ d_1]$
$[d_1 : e_1 \succ e_2;\ d_2 : e_2 \succ e_1]$
$[d_1 : f_1 \succ f_2 \succ f_3;\ d_2 : f_3 \succ f_1 \succ f_2]$

Figure 1. An example CP-net

Example 1 The CP-net in Figure 1 is defined over the variables $\{A, B, C, D, E, F\}$; all variables are binary except for the three-valued F. The decision maker specifies unconditional preference over the values of a (denoted in figure by $a_1 \succ a_2$). However, if $A = a_1$ and $B = b_2$ the decision maker prefers c_2 to c_1 (denoted by $(a_1 \wedge b_2) : c_2 \succ c_1$).

Now consider the CP-net above and the following three outcomes: $\alpha = a_1 b_1 c_1 d_2 e_2 f_2$, $\beta = a_1 b_1 c_2 d_2 e_2 f_2$, and $\gamma = a_1 b_1 c_2 d_1 e_2 f_2$. α and β assign the same values to all variables except C. α assigns to C a value that is preferred to the value β assigns to see given the assignment to the parents of C (denoted $Pa(C)$). Therefore, $\alpha \succ \beta$ is a consequence of this CP-net. The same argument applies to β and γ, with respect to the variable D, and thus, $\beta \succ \gamma$ is a consequence of this CP-net as well. $\alpha \succ \gamma$ cannot be derived directly from the CP-net above. However, this relation can be inferred via transitivity from $\alpha \succ \beta$ and $\beta \succ \gamma$.

In the following examples all variables are binary, although the semantics of both CP-net and TCP-net is defined with respect to arbitrary finite domains.

Example 2 Figure 2(a) illustrates a simple CP-net over three variables A, B, and C; a is unconditionally preferred over $\bar{a}$, and b is unconditional preferred over $\bar{b}$, while the preference relation over the values of C is conditioned on both A and B. The solid lines in Figure 2(c) show the preference relation over outcomes that this CP-net induces. The top element is the worst outcome while the bottom element is the best one. Arrows are directed from less preferred to more preferred outcomes.

[3] Naturally we expect this set $\mathbf{Z}$ to be the minimal context upon which the relative importance between X_i and X_j depends.

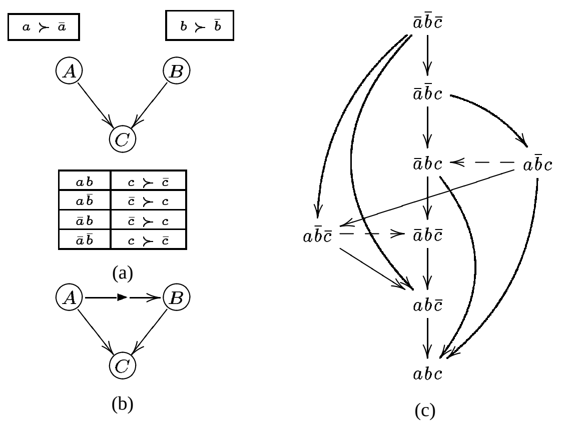


Figure 2. Illustrations for Example 2.

Figure 2(b) displays a TCP-net that extends the CP-net above by adding an i-arc from A to B. Thus, A is absolutely more important than the B. This induces additional relations among outcomes, captured by the dashed lines in Figure 2(c).

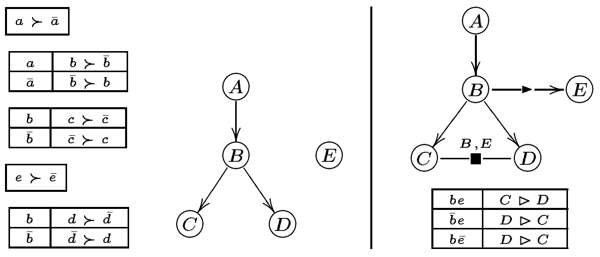


Figure 3. Illustrations for Example 3.

Example 3 Figure 3(a) illustrates a CP-net over five variables A, B, C, D, and E. Figure 3(b) presents a TCP-net that extends this CP-net by adding an i-arc from B to E and a ci-arc between C and D. The relative importance of C and D depends on the assignment to B and E. When B and E are assigned be, then $C \rhd D$. When B and E are assigned $b\bar{e}$ or $\bar{b}e$, then $D \rhd C$. Finally, when B and E are assigned $\bar{b}\bar{e}$, the relative importance between C and D is unspecified. The CIT of this ci-arc is also presented in Figure 3(b).

Depending on the application, a typical process of constructing a TCP-net would commence by asking the decision maker to identify the variables of interest, or by presenting them to the user, if they are fixed. For example, in the application of CP-net to web-page configuration presented in [7], the web-page designer chooses a set of content elements, which correspond to the set of variables. In the case of an online shopper-assistent agent, the variables (e.g., the possible components of a PC) are likely to be fixed. Next, the user is asked to consider for each variable, the value of which other variable influences her preferences over the values of this variable. At this point cp-edges and CPTs will be introduced. Next, the user will be asked to consider relative importance relations, and the i and ci edges will be added. For each ci edge, the corresponding CIT will be filled.

4 PREFERENTIAL CONSTRAINT-BASED OPTIMIZATION

We now discuss the problem of constraint-based optimization given a TCP-net and a set of constraints. We restrict ourselves to a class of *conditionally acyclic* TCP-nets. This class is formally defined in the full paper. Inituitively, conditional acyclicity is an extension of the notion of acyclicity to a graph in which the direction of an edge is conditional on the value of some of the nodes – as in TCP-nets.

Given a TCP-net $\mathcal{N}$ and a partial assignment to its variables, it is simple to determine an outcome consistent with this assignment that is preferentially optimal with respect to $\mathcal{N}$, as shown in [4]. We traverse the variables in some topological order induced by the CP-net part of $\mathcal{N}$ and set each unassigned variable to its most preferred value given its parents' values. The relative importance relations do not play a role in this case.

If some of the TCP-net variables are mutually constrained by a set of hard constraints, $\mathcal{C}$, then determining the set of Pareto-optimal [4] feasible outcomes is not trivial. A branch and bound algorithm for determining such the set of optimal feasible outcomes with respect to an acyclic CP-net was introduced in [3]. This algorithm has the important *anytime* property – once an outcome is added to the current set of non-dominated outcomes, it is never removed. Figure 4 presents a modified version of this algorithm that works with conditionally acyclic TCP-nets and retains the *anytime* property.

The algorithm in Figure 4 uses the topology of the TCP-net to order the variables during the search process. More important variables, i.e., variables that are "higher-up" in the network, are assigned values first. Each variable is assigned the most preferred value in the current context that does not violate the constraints. Thus, by careful selection of variables and values, we ensure that new solutions cannot dominate solutions that were found earlier.

An important element of the algorithm is the test performed in the line before last in the Search routine. There, candidate solutions are compared against the set of current solutions. In [4], this comparison is referred to as *dominance testing*. We discuss the algorithm in more detail, and dominance testing in particular, in the full paper. However, it is important to stress that this element of the algorithm is where constrained optimization and constraint satisfaction differ. However, this difference shows up *only* when (1) more than a single optimal solution is required (because dominance testing is not applied until we generate the first solution), and (2) dominance testing is hard. In CP-nets, [6] shows that when the graph is a polytree, dominance testing is polynomial. In that case, the complexity of the Search procedure is analogous to that of generating all feasible solutions. However, in more complex networks, dominance testing in CP-nets, and therefore, in TCP-nets, is NP-hard.

5 TCP-NETS FOR PRODUCT CONFIGURATION

During the last decade, the *configuration* problem received considerable attention both in academia and industry. Informally, a configuration problem is characterized by two key features [19]:

1. The artifact being configured is assembled from instances of a fixed set of well-defined component types, and
2. Components interact with each other in predefined ways.

[4] An outcome o is said to be Pareto-optimal with respect to some preference order $\succ$ and a set of outcomes S if there is no other o' such that $o' \succ o$.

```
Search ($\mathcal{N}, \mathcal{C}, \mathcal{K}$)
Input: Conditionally acyclic TCP-net $\mathcal{N}$, Constraints $\mathcal{C}$,
        Context $\mathcal{K}$ (partial assignment on some variables of the original TCP-net)
Output: Set of all solutions for $\mathcal{C}$ that are Pareto-optimal w.r.t. $\mathcal{N}$

Choose any variable $X$ s.t. there is no cp-arc $\langle\overrightarrow{Y, X}\rangle$,
  no i-arc $(\overrightarrow{Y, X})$, and no $(X, Y)$ in $\mathcal{N}$.
Let $x_1 \succ \ldots \succ x_k$ be the preference ordering of $\mathcal{D}(X)$
  given the assignment on $P(X)$ in $\mathcal{K}$.
Initialize the set of local results by $\mathcal{R} = \emptyset$
for $(i = 1;\ i \leq k;\ i++)$ do
  $X = x_i$
  Strengthen the constraints $\mathcal{C}$ by $X = x_i$ to obtain $\mathcal{C}_i$
  if $\mathcal{C}_j \subseteq \mathcal{C}_i$ for some $j < i$ or $\mathcal{C}_i$ is inconsistent then
    continue with the next iteration
  else
    Let $\mathcal{K}'$ be the partial assignment induced by $X = x_i$ and $\mathcal{C}_i$
    $\mathcal{N}_i$ = Reduce ($\mathcal{N}$,$\mathcal{K}'$)
    Let $\mathcal{N}_i^1, \ldots, \mathcal{N}_i^m$ be the components of $\mathcal{N}_i$ that are connected
      either by the edges of $\mathcal{N}_i$ or by the constraints $\mathcal{C}_i$.
    for $(j = 1;\ j \leq m;\ j++)$ do
      $\mathcal{R}_i^j$ = Search ($\mathcal{N}_i^j, \mathcal{K} \cup \mathcal{K}', \mathcal{C}_i$)
    if $\mathcal{R}_i^j \neq \emptyset$ for all $j \leq m$ then
      foreach $o \in \mathcal{K}' \times \mathcal{R}_i^1 \times \cdots \times \mathcal{R}_i^m$ do
        if for each $o' \in \mathcal{R}$ holds $\mathcal{K} \cdot o' \not\succ \mathcal{K} \cdot o$ then Add $o$ to $\mathcal{R}$
return $\mathcal{R}$

Reduce ($\mathcal{N}, \mathcal{K}'$)
foreach $\{X = x_i\} \in \mathcal{K}'$ do
  foreach cp-arc $\langle\overrightarrow{X, Y}\rangle \in \mathcal{N}$ do
    Restrict the CPT of $Y$ to the rows dictated by $X = x_i$.
  foreach ci-arc $\gamma = (Y_1, Y_2) \in \mathcal{N}$ s.t. $X \in \mathcal{S}(\gamma)$ do
    Restrict the CIT of $\gamma$ to the rows dictated by $X = x_i$.
    if, given the restricted CIT of $\gamma$, relative importance
      between $Y_1$ and $Y_2$ is independent of $\mathcal{S}(\gamma)$, then
        if CIT of $\gamma$ is not empty then
          Replace $\gamma$ by the corresponding i-arc.
        else Remove $\gamma$.
  Remove from $\mathcal{N}$ all the edges involving $X$. return $\mathcal{N}$.
```

Figure 4. Search Algorithm for Non-dominated Outcomes

Therefore, selecting and arranging combinations of parts that satisfy given specifications form a core of the configuration task. While there has been a wide and growing research in modeling configuration problem, and efficient problem solving methods, there is still a need for more work on modeling and learning user preferences, and using these to achieve configurations that are not only feasible, but also satisfactory from the user point of view. The necessity of these issues is emphasized by almost every paper on configuration, e.g. [10, 11, 13, 15, 21], especially when high-level configurators [11] for particular, real-life domains are discussed. The importance of seeing user preferences as a part of the configuration problem mostly entailed from the fact that it is often the case that many configuration problems, e.g. product configuration, are weakly constrained and have numerous solutions. The value of these solutions, from the subjective point of view of a particular user, may vary significantly between the solutions.

Recently, the issue of user preferences as a part of the configuration problem has received considerable attention in the configuration community. Modeling tradeoffs as additional constraints on the configuration problem was addressed in [10]. Preference-based search that is guided to preferred solutions was introduced in [14]. A preference programming framework which is based on a logical approach for treating preferences was proposed in [15]. In addition, various approaches for modeling and dealing with soft constraints as a part of the constraint satisfaction process have been closely examined in the CSP community [2]. However, the latter approaches address over-constrained problems that should be relaxed. They incorporate preference information by attaching truth values to *constraints*, not to the variables or variable values. Therefore, this area is mainly about optimization *of* partial constraint satisfaction, and not about solution optimization *in face* of constraint satisfaction.

We believe that the TCP-net model provides a good, and in many respects unique, basis for a preference-based configuration framework because of the following reasons:

1. The development of the CP-net, and subsequently of the TCP-net models, was guided by the naturalness of the represented statements. Therefore, the preferential statements that are captured by TCP-nets are likely to be found intuitive for users at all levels. In particular, we argue that all the preferential statements that were discussed in [15] can be captured by a TCP-net.
2. The TCP-net model is graphical, and its structure is induced by the assumptions of preferential *independence* between the problem variables. This structure can be utilized in developing specialized algorithms that exploits various properties of this structure [6]. Note that exploiting advantages of graphical, independence-based representation models is widely accepted in AI, and in particular in utility representation [1, 17], constraint satisfaction [22], and probabilistic reasoning [18].
3. The TCP-net model has a well-defined semantics in terms of qualitative decision theory. This allows for a clear and clean analysis of its expressiveness and its computational properties with respect to inference, consistency, etc.
4. The branch and bound CSP algorithm that is guided by a TCP-net has two important properties: First, this algorithm is anytime, i.e., once a solution is added to the generated set of Pareto-optimal solutions, it is never removed from this set. Second, if we can be satisfied by one Pareto-optimal solution, then its generation is not harder than the generation of a single feasible solution. In general, these properties are rare in the world of constraint optimization.

In what follows, we address some additional issues that may be found important in the context of preference-based configuration.

First, in decision theory, the preference order reflects the preferences of a decision maker. The typical decision maker in preference-based product configuration is the consumer. However, in some domains, the product vendor may decide that customer's preference elicitation is inappropriate, or simply impossible. In this case, the role of the decision maker may be relegated to a *product expert*, since she is likely to have considerable knowledge about appropriate component combinations. Given some information about the customer, such a product expert can argue about the expected preferences of this customer. Following an example in [15], in case of a car configuration, the expert may determine that if the customer is young and a male then he is likely to prefer white cars on red cars, or that the fashioned look is a more important parameter than reliability. In this case, the product expert may specify a TCP-net over the variables that stand for parameters of both the product and the customer. The latter variables will serve as *network parameters*, and will be instantiated at the beginning of each session of personalized configuration, given some personal information about the customer. Clearly, each such variable will be a root variable in the TCP-net – it will not participate neither in ci-arcs nor in i-arcs, and it will have only outgoing cp-arcs. However, it may serve as a selector variable in some ci-arcs of the network.

Note that relegation of the role of the decision maker from the customer may be done because of various reasons. For example, [7] presents a CP-net based approach for configuration of presentations, which is illustrated on web pages. In this approach, the process of interactive document presentation is viewed as a configuration problem whose goal is to determine the optimal document appearance while taking into account the preferences of the document's author, e.g. an editor of an online newspaper, and viewer interaction with the document.

An additional issue that, at least in some cases, can be addressed naturally using the TCP-net model, is *non-rigidness* of the variable set [16, 19, 20, 23]. As it is argued in [19], different components for the same functional role may need nonidentical sets of additional components, or functional roles. In this case, the set of variables in a solution changes dynamically on the basis of assignments of values to other variables. Now, consider a variable X that, for instance, stands for a particular functional role. Observe that nothing in the semantics of the TCP-net prevents one of the values in the domain of X to stand for the *absence of* X. This way, the presence/absence of some variables may condition the presence of some other variables, their relative importance, and the preference on their values. Note, that the former conditionings are entailed by the activity constraints on the variables, which are defined by the core configuration problem, and not by the preferences of a user. In this case, they can be added into the TCP-net automatically, *after* the preference elicitation stage, by a straightforward extension of the specified TCP-net. Such an approach of automatic extension of CP-nets was exploited in [8] for preference-based presentation of tree-structured multimedia documents. In this domain, (i) the preferences on the value (= option of content appearance) of a document's component may be conditioned by the value of some other components, and (ii) whether it should be shown or hiden depends directly on which of its ancestors are shown and which are hidden.

Note that we are not claiming that this approach will be feasible and/or natural for any dynamic preference-based configuration task. However, if it will, then one of the benefits seems to be the fact that this task could be treated exactly as it was static, using the Search algorithm (see Figure 4). Of course, if the variable set of the problem is very dynamic, then the complexity of an automatically extended TCP-net may be high, and thus efficient consistency verification of the specified preference relation is important. For discussions and results on this issue we refer our reader to [6, 5].

Another issue of the configuration process in which the TCP-net model seems to be beneficial, is *explanation generation* for users [9]. The goal of an explanation for a generated configuration is to help the user understand the decisions that were taken during the search process, e.g. why this particular configuration was chosen. The importance of explanation is even more significant if the configuration task is preference-based, i.e., it is about optimization, not only satisfaction. It seems reasonable to build an explanation in respect to the values of the variables in the accepted solution according to the same order that these variables were examined during the search. Recall that the flow of the Search algorithm in Figure 4 is *guided by the structure* of the TCP-net. If so, then this structure may be exploited during the explanation generation. However, this issue is not in the scope of this paper, and we leave it as a possible direction for future research.

One of the interesting applicative issue in the framework of elicitation of qualitative preferences is model acquisition from speech/text in natural language [12]. Observe that the intuitiveness of the qualitative preferential statements is highly related to the fact that, at least most of them, have a straightforward representation in natural language of everyday life. In addition, collections of preferential statements seems to form a domain that is apriori constrained in a very special manner. This may allow us to develop specialized techniques and tools for understanding the corresponding language. Finally, both offline and online language understanding should be considered, since a user can either describe her preferences offline, as a self-contained text, or can be asked online, as a part of interactive process of (possibly mixed) preference elicitation and preference-based constraint optimization [3].

REFERENCES

[1] F. Bacchus and A. Grove, 'Graphical Models for Preference and Utility', in *Proc. of UAI-95*, pp. 3–10, (1995).

[2] S. Bistarelli, H.Fargier, U. Montanari, F. Rossi, Thomas Schiex, and Gerard Verfaillie, 'Semiring-Based CSPs and Valued CSPs: Frameworks, Properties, and Comparison', *Constraints*, **4**(3), 275–316, (September 1999).

[3] 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).

[4] C. Boutilier, R. Brafman, H. Hoos, and D. Poole, 'Reasoning with Conditional Ceteris Paribus Preference Statements', in *Proc. of UAI-99*, pp. 71–80, (1999).

[5] R. Brafman and C. Domshlak, 'Introducing Variable Importance Tradeoffs into CP-Nets', in *Proc. of UAI-02*, (2002).

[6] C. Domshlak and R. Brafman, 'CP-nets - Reasoning and Consistency Testing', in *Proc. of KR-02*, (2002).

[7] C. Domshlak, R. Brafman, and S. E. Shimony, 'Preference-based Configuration of Web Page Content', in *Proc. of IJCAI-01*, pp. 1451–1456, (2001).

[8] C. Domshlak and S. E. Shimony, 'Predicting Likely Components in CP-net based Multimedia Systems', Technical Report CS-01-09, Dept. of Computer Science, Ben-Gurion Univ., (2001).

[9] E. Freuder, C. Likitvivatanavong, and R. Wallace, 'Explanation and Implication for Configuration Problems', in *Proceedings of 4th Workshop on Configuration (IJCAI-01)*, pp. 31–37, Seattle, US, (August 2001).

[10] E. Freuder and B. O'Sullivan, 'Modeling and Generating Tradeoffs for Constraint-Based Configuration', in *Workshop on Configuration (IJCAI-01)*, (2001).

[11] A. Haag, 'Sales Configuration in Business Processes', *IEEE Intelligent Systems and their Appl.*, **13**(4), 78–85, (1998).

[12] G. James, 'Challenges for Spoken Dialogue Systems', in *Proceedings of the IEEE ASRU Workshop*, (1999).

[13] U. Junker, 'A Cumulative-Model Semantics for Dynamic Preferences on Assumptions', in *Proc. of IJCAI-97*, pp. 162–167, Nagoya, Japan, (August 1997).

[14] U. Junker, 'Preference-based Search for Scheduling', in *Proceedings of Seventeenth National Conference on Artificial Intelligence*, pp. 904–909. AAAI Press, (August 2000).

[15] U. Junker, 'Preference Programming for Configuration', in *Workshop on Configuration (IJCAI-01)*, pp. 50–56, Seattle, US, (August 2001).

[16] S. Mittal and B. Falkenhainer, 'Dynamic Constraint Satisfaction Problem', in *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 25–32. AAAI Press, (1990).

[17] P. La Mura and Y. Shoham, 'Expected Utility Networks', in *Proc. of UAI-99*, (1999).

[18] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, San Mateo, CA, 1988.

[19] D. Sabin and R. Weigel, 'Product Conguration Frameworks - A Survey', *IEEE Intelligent Systems and their Appl.*, **13**(4), 42–49, (1998).

[20] T. Soininen and E. M. Gelle, 'Dynamic Constraint Satisfaction in Conguration', in *Proceedings of AAAI Workshop on Conguration*, (1999).

[21] T. Soininen and I. Niemelä, 'Formalizing Configuration Knowledge Using Rules with Choices', in *Int. Workshop on Nonmonotonic Reasoning*, (1998).

[22] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[23] M. Veron and Aldanondo, 'Yet Another Approach to CCSP for Configuration Problem', in *Proceedings of 3rd Workshop on Configuration (ECAI-00)*, Berlin, Germany, (August 2000).

Neural networks to approximate data collection or computer code in constraint based design applications

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

Abstract. During a design or configuration process, some knowledge cannot directly be provided as analytical relations between design variables. In such cases, an analytical approximated relation has to be built.

The work is focused on the approximation capabilities of different kind of neural networks and the integration of the corresponding approximations into a constraint based system, such as constraint logic programming.

1 INTRODUCTION

Building an application on top of constraint programming requires that all constraints of the problem have to be made explicit using a set of basic constraints provided by the language. But, in some cases, the analytic relation between the variables is not available. The reasons for that unavailability can be the complexity of the underlying model, or the experimental nature of the relation or even confidentiality.

In such situations, embedding these relations within a constraint programming environment is not straightforward. The work under progress is aimed at studying how approximation schemes coming from neural networks may be used to build constraints when the analytical formulation of the underlying relation is not available. The first part of the paper precises the type of relations we intend to deal with. The second part is centered on neural networks approximation capability. And the third part presents some preliminary results.

2 NON ANALYTIC KNOWLEDGE

When dealing with the design of complex systems, such as aircrafts, two types of knowledge, not available as analytical relations, may be encountered : collections of data and existing computer code.

2.1 Data collection

In this case, basic knowledge about the relation is made of sets of points where the relation is satisfied. Typically, these points represent a family of curves expressing the relation between several variables. For instance, relations between altitude, speed and thrust for a given turbojet engine are available under this form. They are provided by the engine manufacturer and result from thermodynamics theory, simulation and experimental validation. When trying to take into account engine performances into a design process, you have to find a way to integrate such data.

Another example of such knowledge is the use of statistical data : the data set represents in fact the value of different parameters for a set of existing items, which are assumed to be extrapolated. For instance, in an aircraft, the repartition of weights for each subsystems is often computed by considering existing similar aircrafts.

2.2 Computer object code

In this case, knowledge about the relation between variables is provided as an existing computer object code (procedure). Several reasons may explain why translating this procedure into a set of basic constraints is not possible :

- confidentiality : when design requires cooperation between different entities, sometimes you are allowed to use a function, but not to know how it is built. In this case typically, an object code like a shared object library and a description of the basic call for the different functions are the only knowledge source;
- translation : even by assuming that the source code is available, the lack of documentation or the inherent complexity of the procedure like simulation-based procedures for instance, makes the translation into constraints hard.
- duration : in some cases, the difficulty arise from procedure running time execution; the approximation is needed to reduce computing time and to allow the integration of the knowledge within a constraint propagation mechanism.

2.3 Proposed framework

When dealing with design applications, such situations are frequent. In aeronautic design applications based on optimization techniques [16], approaches based on approximation techniques have been proposed. A distinction is made between inputs and ouputs and the approximation is always to compute outputs when the inputs are given.

As the use of constraint programming for design is more recent, these situations have received less attention. Contrary to classical programming techniques, in constraint programming the user only express the relation between variables (constraints) and do not impose a particular data flow. A computation procedure (constraints propagation), embedded in the constraint programming language computes domains reduction for the variables and decides, at each propagation step, of the data flow within a constraint.

2.3.1 Generalization

Although different in nature, both situations are very close from the approximation point of view. The distinction relies in the definition of the learning space which can be used to build the approximation :

- In the data collection case, the space is given explicitly (the given set of points);

[1] ONERA-DCSD, BP 4025, 2 av E. Belin, 31055 Toulouse CEDEX 4, France {eric.bensana,taufiq.mulyanto@cert.fr}

- In the computer code case the limits of the space for the variables are given and thus every point in this subspace can be used to build the approximation.

The study is focused on the approximation of real functions of the type : $Y = f(X_1, X_2, ..., X_n)$. Actually, we consider only functions where the X_i and Y are reals (but in the future, integers should also be considered). The aim is to build a constraint of arity $n+1$, approximating f : $AC_f(X_1, X_2, ..., X_n, Y)$. To allow the reuse of this approximating constraint in the design application, AC_f should be expressed using a set of basic constraints available in the constraint programming environment.

In case of multiple outputs, like for example in computer code, a relation $\{Y_1, .., Y_k\} = f(X_1, ..., X_n)$ can be split into k basic relations $Y_i = f_i(X_1, ..., X_n)$, leading to the building of k approximations. By doing this, we limit possible interferences in the approximation building process. But this decomposition is only a possibility, if relations between outputs are required the full relation should be taken into account.

2.3.2 *Knowledge characterization*

In order to characterize the initial knowledge, we will assume that the following information about the relation is provided :

- a characterization of the variables X_i and Y : type (input or output) and domain of values;
- the definition of the computation support : either a set of data points where the relation is known $(X_1^i, X_2^i, ..., X_K^i, Y^i)$ or an executable computer code F implementing f.

We assume also that the approximating constraint building process is done before the use of the resulting constraint in the design application.

2.4 Approximation background

Approximation is widely used in engineering and specially in design applications. Starting from a set of data representing an unknown function f, the aim is to determine a function g which is as close as possible to the training data set, according to a criteria.

Interpolation is a special case of approximation where the function g must go through all the data points of the training set. Several interpolation methods exists like Lagrange, Neville-Aitken, Newton or cubic spline. They differ mainly by the type of basic functions used to build the interpolating function.

Different approaches have been developped for building approximations :

Least Square : approximations according to least-square optimization like polynomial, exponential, power, trigonometric logarithmic approximation [4, 19] are, because of their simplicity, certainly the most widely used techniques. Once a general form for the approximation is selected, a least-square optimization process finds the parameters of the function g.

Response Surface : originally developped to analyze the results of physical experiments, this method postulates that f can be approximated by $g(x) = h(x) + \varepsilon$ where h is a polynomial function (usually linear or quadratic) and ε a random error of mean 0, independent of observations. The parameters of h are computed using least square regression techniques which minimize the sum of squares of deviation between predicted values and actual values.

Kriging methods : these methods, named from Krige, were originally defined to cope with stochastic aspects in data analysis in the mining domain. Kriging approximation [14, 5, 8], introduces also a stochastic component in the approximating function. g is defined as $g(x) = p(x) + Z(x)$ where p is usually a polynomial function (very often restricted to a constant term) which takes into account the global approximation of f and Z is the realization of a stochastic process of mean 0, variance σ^2 and non-zero covariance which approximates the local deviation. Depending on the choice of the correlation function kriging methods can be used to build interpolations or approximations.

Neural networks : multi-layered perceptron [20, 18, 11, 12], radial basis functions neural nets [10], self-organizing maps [2] or neural gas [21] are known to have good approximating properties and will be discussed in section 3.

Genetic algorithms : chromosoms represent trees of computations where each gene represents a basic unary or binary arithmetic functions. By applying crossing and mutation, new functions can be built which are matched again the learning set to evaluate their fitness [1, 3].

Fuzzy rules : the function f is approximated by a set of fuzzy rules like $if\ X_1\ is\ A_1\ and...and\ X_n\ is\ A_n\ then\ Y\ is\ B$, where A_i and B are fuzzy sets [7, 13]. For example, in [9], rules can be derived from the analytical knowledge of f and its first derivative. This approach is well adapted for example to introduce analytical rules into fuzzy controllers.

Design of experiments : Although the definition of experimental plans is not an approximation technique, it is often associated to an approximation scheme in order to limit the identification, training or learning preliminary phase by avoiding extensive computations. These approaches have first been defined by the english statistician Fisher in the 20s and they have been highlighted in the 80s by Taguchi in the domain of production quality [17].

2.5 Work orientation

The goal is to define a generic approach for embedding both data collection and computer code compatible with constraint programming and which does not require too much user interaction. By considering some of the specificities of the foreseen applications, the domain of investigation can be restricted :

- stochastic aspects are not present in the case of computer code, so Response Surface, including stochastic aspects will be discarded;
- kriging methods, also including stochastic aspects, can be used in computer code case, like for instance in Design and Analysis of Computer Experiments (DACE) [15], but they have not been considered here because of their complexity;
- interpolation is not required because we consider design applications at a high level of description, which imply imprecision and because it is incompatible within the computer code case.
- before using least-square approximation, you need to look at data before selecting the type of approximation to use or build different approximations before selecting the one which matches best : this point does not match our goal of low level interactivity

- fuzzy approximation does not match the fact that the approximation should be directly translated into a set of basic arithmetic constraints;
- genetic approximation allows to build complex functions by combining a set of basic functions but this approach seems to be quite complex to implement;
- design of experiments methods will be needed when adressing the computer code case to help to generate an explicit training set.

These reasons lead us to focus on neural networks, even if other methods like those derived from kriging methods could be used.

3 NEURAL NETWORKS

3.1 Mathematical formulation for neural nets

Because of the numerous types of neural networks (NN), we will limit our presentation to the feed-forward types, which are those used for approximation. A neuron model is given in figure 1. Information is propagated through it according to the arrows.

Inputs are represented by the vector $E = \{e_1, e_2, \ldots, e_n\}$ and $W = \{w_1, w_2, \ldots, w_n\}$ is the vector of the weights of the vertices linking inputs to the neuron. The neuron is modelled by an input integration function $g(.)$ and a transfer function $f(.)$. These functions allow to compute the output of the neuron s. The output vertex allows to propagate the output to others neurons. The bias parameter b helps to improve the model.

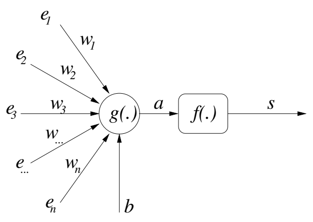


Figure 1. Mathematical model of a neuron

The function $a = g(W, E, b)$ combines the inputs, the weights and the bias in a single value. The transfer function (also called activation function) takes into account non-linear aspect.

3.2 Definition of a neural network

The definition of a given neural network is usually made in two steps :

1. the definition of the general architecture of the network : number of layers, number of neurons, types of the functions f and g;
2. the training of the network, in order to compute the values of the different parameters : weights and biases.

There is no real methodology for deciding of the general architecture of a network for a given problem. Thus intuition or experience are often the only rules. As far as approximation is considered, several types of neural networks architectures are known to be efficient : multi-layered perceptron (MPL), radial basis function neural nets (RBFNN), self-organizing maps (SOM) or neural-gas (NG).

3.3 Neural nets for approximation

In this paper, only MLPs and RBFNNs are considered. Other NN architectures like SOMs or NGs should be considered later in the study.

3.3.1 *Multi-Layers Perceptron*

The multi-layers perceptron (MLP) [20, 18, 11, 12] is one of the most simple network to build. Their basic characteristics are :

- they are structured by layers : one input layer, several hidden layers and an output layer;
- inside a layer a neuron is not connected to any other neuron of the layer but connected to all neurons of previous and next layer;
- the input layer has one neuron for each input and can be forgotten by considering that each input is connected to all neurons of the first hidden layer : the number of layers is the number on hidden layers + 1
- the activation function g is common to all neurons of the hidden layers.;
- for each neuron of a hidden layer :
  - the integration function g should be a summation $\sum_i e_i.w_i + b$,
  - the transfer function f must be monotonic increasing like for instance the sigmoid function $1/(1 + exp(-\alpha.u))$ or the hyperbolic tangent $tanh(\alpha.u)$
- for each neuron of the output layer :
  - the integration function is also a summation;
  - the transfer function is linear;

Consequently, when two layers MLP are considered (one hidden layer and the output layer), each output neuron computes the following function :

$$y = \sum_{j=1}^{p} b_j.g(\sum_{k=1}^{q} a_{j,k}.x_k + c_j)$$

where :

- p is the number or neurons of the hidden layer and q is the number of inputs;
- g is the activation function associated to the neurons of the hidden layer;
- the b_j are the weights of the links between the neurons of the hidden layer and the neuron of the output layer considered;
- the $a_{i,j}$ are the weights of the links connecting the inputs to the neurons of the hidden layer;
- c_j is the bias associated to the j^{th} neuron of the hidden layer;
- the x_k are the inputs and y is the output.

The training is supervised, meaning that a training set has to be determined. The learning algorithms which allows to compute the weights of the network are generally error backpropagation algorithms and the modification of the weights is based on the quadratic error.

3.3.2 *Radial Basis Functions Neural Nets*

Radial Basis Function Neural Nets [10] are very close to the multi-layered perceptron. Their structure is also layered, but there is only one hidden layer. The neurons of the hidden layer use a regular and radial symetric activation function, which is assumed to deal with proximity between data. A regular, radial symetric function is of the form : $g(\vec{x}) = K(\|\vec{x}-\vec{c}\|)$ where :

- $\vec{c}$ is the class center
- $\|$ is a norm : for instance euclidian distance $\sqrt{\sum_i (x_i - c_i)^2})$
- K is a Green function like for example the e^{-ax^2} with $a > 0$;

For instance, gaussians are regular radial symetric functions.

The neurons of the output layer, as in the MLP case, perform a linear combination of the output of the neuron of the hidden layer. The approximation performed by each neuron of the output layer is then :

$$\sum_{j=1}^{p} b_j . g(\|\vec{x}-\vec{c}_i\|, \sigma_i)$$

Three types of training are available for this type of network :

- a one step algorithm which computes directly the networks parameters by minimizing the approximation error;
- two steps algorithm :
  1. the learning of the characteristics of the radial basis functions : one neuron for each training point or one neuron for each class of input (which may require a preliminary classification of the training set);
  2. learning with some kind of backpropagation algorithms of the weights b_j of the links between the neurons of the hidden layer and those of the output layer.
- a training of the output layer only : the hidden layer is fixed by the user or randomly and the learning reduces to the solving of an linear optimization problem.

Generally this type of network is easier to build than the MLP.

3.3.3 *Adequation*

These two types of neural networks, have *a priori* interesting properties :

- their approximating capabilities are known;
- they have a rather simple structure : simpler for the RBFNNs than for the MLPs;
- the approximation formulation can be easily and directly translated as a set of basic arithmetic constraints : sum, square, exponentiation, product *etc.*
- they can deal with multiple outputs models.

4 EXPERIMENTATION

The purpose of the ongoing experimentations is to test different neural networks architectures to evaluate their pertinence for building approximating constraints. In order to evaluate and compare different architecture, evaluation criteria and test cases must be stated.

4.1 Evaluating neural-based approximation

The training of neural networks is usually done using to set of examples : a training set which is used by the learning algorithm to computes weights and biases and an evaluation set, different from the training set, which is used to evaluate how the neural network really performs.

To evaluate, the pertinence of a neural-based approximation, two aspects must be balanced :

- the approximation quality which measures the intrinsic performance of the approximation built *w.r.t* the initial knowledge; this evaluation is done by computing both approximation error for the points in the training set and the generalization error for the points of the evaluation set; both errors are related to the size of the network in term of number of neurons, the approximation error decreases when the size increases, but if the size becomes too high the generalization error may increases;
- the propagation effectiveness which measures how constraint propagation performs on the approximating constraint built from the neural net; propagation effectiveness usually decreases when the number of neurons increases.

To improve propagation effectiveness and avoid tow large generalization error, it seems interesting to build neural networks as small as possible.

Another aspect, which can taken into account, is related to running time : learning time or propagation time.

4.2 First experimentations

Preliminary experimentations have been conducted, using the Neural Network Toolbox of Matlab.

4.2.1 *Example*

An example has been built from the following function (see figure 2) :

$$z = [y \cdot exp^{-(0.5x-1)^2} \cdot sin(2y)] + \frac{y(1+x)^2}{20}$$

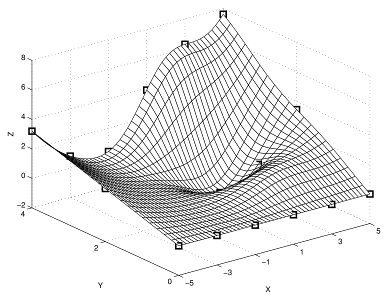


Figure 2. Surface for z=f(x,y)

A training set of 55 points has been built by varying x from -5 to 5 (step 1) and y from 0 to 4 (step 1) and an evaluation set of 14 points have been built by varying x from -4.5 to 4.5 (step 1) and y from 0.5 to 3.5 (step 1). Consequently, $z = f(x,y)$ belongs to the interval $[-0.7788, 7.6171]$. Globally the data set is formed of 69 triples corresponding to the values (x, y, z).

4.2.2 Architecture of networks

The following architectures have been tested and compared :

- one hidden layer MLP (MLP1) with 3,5,7 or 9 neurons in the hidden layer
- two hidden layers MLP (MLP2) with a neuron repartition 2-1, 2-3, 2-5, 2-7, 1-2, 3-2, 5-2, and 7-2 (x-y meaning that there are x neurons in the first layer and y in the second layer)
- RBFNN with 3,5,7 or 9 neurons in the only hidden layer;

All architectures have two inputs and one output.

4.2.3 Learning algorithms

As sixteens different learning algorithms are available in the Matlab ToolBox with their own parameters, the testing has been limited.

Three backpropagation algorithms have been selected for MLPs : Levenberg-Marquardt, Bayesian Regularization and Gradient Descent with momentum and adaptative learning rate.

For RBFNNs, the only algorithms available are of type one-step : there is no real learning algorithm, in fact the building procedures directly computes the architecture and the values of the parameters from the training set. As inputs are read, neurons are added to the hidden layer and parameters adjusted to limit the error to a maximal value. The only parameters are the maximal error and a spread value, which allows to control the smoothness of the approximation built. The bias for the neurons is computed from the spread.

4.2.4 Testing

As learning implies random aspects, for each couple (network architecture, learning algorithm), hundred runs have made in order to compute mean and variance for the different criteria.

Propagation effectiveness tests have been done using the constraint logic programming language Prolog IV. For a trained network, the formulation of the computation it performs is translated into predicates using basic constraints of the language.

4.2.5 Preliminary results

On the simple example, both approximation quality and propagation effectiveness have been measured for the sixteen different configurations of NNs and with the learning algorithms selected. Detailed results will not be presented here, but only the main lessons learned :

- to get a same approximation quality, an MLP needs a smaller number of neurons than a RBFNN;
- the most appropriate algorithm for MLP is Bayesian Regularization;
- for two hidden layers MLPs, architectures of type N-2 give better results than those of type 2-N;
- for MLP and for a same number of neuron, the one hidden layer architecture gives better results than the two hidden layers one;
- for one hidden layer architectures and for a same number of neurons, MLPs with one hidden layer perform better than RBFNNs;
- propagation effectiveness degrades in quality and running time when the number of neurons increases;
- for networks with only one hidden layer (MLP1 or RFBNN), propagation effectiveness is similar but the MLP formulation is slower.

To summarize, simplest architectures must be preferred like MLP with one hidden layer or RBFNN. At this point a final choice is difficult : on one hand, the building of RBFNN is simpler and can be done automatically, but on the other hand MLP need less neurons than RBFNN to achieve the same quality and thus are better for constraint programming translation. Further experiments are required before deciding of the best architecture.

5 CONCLUSION

This work, partially supported by SPAE, is under progress and thus only limited and partial results are presently available. Several remaining points can be addressed :

- more extensive validation to compare MLP1 and RBFNN architectures;
- determination of the training and evaluating sets specially in the case of computer code;
- limitations of the approach in terms of total number of variables (inputs and outputs) which can be handled efficiently;
- improvement propagation for the approximating constraint by using more powerfull consistency techniques like box-consistency [6] or defining specific ones;
- impact of considering variables restricted to be integer;
- testing other types of neural networks such as SOMs or NGs or other approximation schemes like DACE and comparison of results;
- applicabilition to real-life design application in the aeronautic area such as radar characteristics or engine performances where altitude, mach number and thrust or consumption are linked.

REFERENCES

[1] L. F. Alvarez, V. V. Toporov, D. C. Hughes, and A. F. Ashour, 'Approximation model building using genetic programming methodology : applications', in *2nd ISSMO/AIAA Internet Conference on Approximations and Fast Reanalysis in engineering optimization*, (2000).

[2] M. Aupetit, P. Couturier, and P. Massotte, 'Function approximation with continuous self-organizing maps using neighboring influence interpolation', in *Neural Computation B*, (May 2000).

[3] J. Frohlich and C. Hafner. Extended and generalized genetic programming for function analysis. Internet.

[4] A. A. Giunta, 'Aircraft multidisciplinary design optimization using design of experiments theory and response surface modelling methods', Technical Report 97-05-01, Multidisciplinary Analysis and Design Center, Virginia, (1997).

[5] A. A. Giunta and L. T. Watson, 'A comparison of approximation modeling techniques : polynomial versus interpolating models', in *7th Symposium on multidiscplinary Analysis and Optimization*. AIAA/NASA/UASF/ISSMO, (sep 1998).

[6] Pascal V. Hentenryck, David McAllester, and Deepak Kapur, 'Solving Polynomial Systems using a Branch and Prune Approach', *SIAM J. Numerical Analysis*, **34**, 797–827, (avr 1997).

[7] B. Kosko, 'Fuzzy systems as universal approximators', *IEEE transactions on computers*, **43**(11), 1329–1333, (1994).

[8] A. Limaiem and H. A. ElMaraghy, 'Curve and surface modelling with uncertainties using dual Kriging', *Journal of Mechanical Design*, **121**, 249–255, (jun 1999).

[9] D. Lisin and M. A. Gennert, 'Optimal function approximation using fuzzy rules', in *International Conference of the North American Fuzzy Information Processing Socier*, (1999).

[10] C. G. Looney, 'Radial basis functional link nets as learning fuzzy systems', Cs479, University of Nevada, Department of Computer Science, (1996).

[11] P. G. Maghami and D. W. Sparks, 'Design of neural networks for fast convergence and accuracy', in *39th Conference on Structures, Structural Dynamics and Materials*. AIAA/ASME/ASCE/AHS/ASC, (sep 1998).

[12] V. Maiorov and A. Pinkus, 'Lower bounds for approximation by MLP neural networks', *Neurocomputing*, **25**, 81–91, (1999).

[13] S. Mitaim and B. Kosko, 'What is the best shape for a fuzzy set in function approximation', in *5th IEEE International Conference on Fuzzy Systemes*, (sep 1996).

[14] T. W. Simpson, J. J. Korte, T. M. Mauery, and F. Mistree, 'Comparison of response surface and kriging models for multidisciplinary design optimization', in *7th Symposium on multidiscplinary Analysis and Optimization*. AIAA/NASA/UASF/ISSMO, (sep 1998).

[15] T. W. Simpson, J. Peplinski, P. N. Koch, and J. K. Allen, 'On the use of statistics in design and the implications for deterministic computer experiments', in *ASME Design Engineering Technical Conference*, (sep 1997).

[16] J. Sobieszczanski-Sobieski and R. T. Haftka, 'Multidisciplinary aerospace design optimization ; survey of recent developments', in *34th Aerospace Sciences Meeting and Exhibit*, (jan 1996).

[17] P. Souvay, *Les plans d'expérience : méthode Taguchi*, AFNOR, 1995.

[18] D. W. Sparks and P. G. Maghami, 'Neural networks for rapid design and analysis', in *39th Conference on Structures, Structural Dynamics and Materials*. AIAA/ASME/ASCE/AHS/ASC, (sep 1998).

[19] R. Unal, R. A. Lepsch, and M. L. McMillin, 'Response surface model building and multidisciplinary optimization using D-optimal design', in *7th Symposium on multidiscplinary Analysis and Optimization*. AIAA/NASA/UASF/ISSMO, (sep 1998).

[20] V. Vysniauskas, C. A. Groen, and B. J. A. Krose, 'The optimal number of learning samples and hidden units in function approximation with a feedforward network', Technical Report CS-93-15, University of Amsterdam, Faculty of computer science and mathematics, (1993).

[21] M. Winter, G. Metta, and G. Sandini, 'Neural-gas for function approximation : a heuristic for minimizing the local estimation error', *IEEE*, (2000).

Representing Software Product Family Architectures Using a Configuration Ontology

Timo Asikainen[1], Timo Soininen[1] and Tomi Männistö[1]

Abstract. In this paper, we study the possibility of applying techniques developed for configuring mechanical and electronics products to configuring software. We analyze and compare at the conceptual level software architecture description languages and configuration modelling concepts. Based on the analysis we are able to define a way of representing much of the architectural knowledge using the configuration modelling concepts. This indicates that it is relatively easy to provide software configuration support using the existing techniques if the software is represented through architectural descriptions. However, there are also some differences that require extending the current conceptualizations of configuration knowledge to capture software products adequately.

1 INTRODUCTION

In the recent years there has been an increasing research effort dedicated to providing better configuration modelling languages and tools. However, the research on configuration has mainly dealt with mechanical and electronics products. At the same time, software product lines or families have become increasingly important in the software industry [1]. The most systematic of such families closely resemble configurable products in that they are composed of standard re-usable assets and have a predefined architecture [1]. A major effort in the software product family and architecture research has been spent on developing architecture description languages (ADLs) for representing these re-usable assets and software architectures. Thus product families and ADLs are natural counterparts in the software domain for configurable products and configuration modelling languages. There are many ADLs and large differences between them [2,3].

In this paper, we study the possibility of applying techniques developed for modelling and configuring mechanical and electronics products to configuring software. A prerequisite for coming up with a general solution to this problem is to define a mapping from the conceptualization of software systems to a conceptualization of configuration knowledge. Towards this end, we analyze three prominent ADLs at the conceptual level and compare them with the major concepts used for modelling configuration knowledge. Based on the analysis and comparison, we show how to represent main concepts of ADLs using the configuration modelling concepts. In addition, we identify several potential needs for extending the configuration modelling concepts with ADL derived concepts.

For the purposes of this paper we concentrate on three important ADLs: Acme [4,5,6], Wright [7,8] and Koala [9,10]. Out of these, Acme has been designed to include features of other ADLs that its designers considered central. The relevance of Acme is further promoted by the fact that one of the goals of Acme is to serve as an interchange language for other ADLs. Wright is a widely cited ADL that has a rigorous semantics and describes behavioural aspects of software. Both the use of formal methods and description of behaviour make Wright important among ADLs. Koala is in commercial use at Philips Consumer Electronics. Being one of the few ADLs used in commercial applications, it is an important example of the practical aspects of ADLs.

As the reference point in the comparison we employ a configuration ontology presented by Soininen et al. [11]. This ontology synthesizes prior conceptualizations of configuration knowledge. Moreover, it is very similar to another recognized configuration ontology presented by Felfernig et al. [12]. Thus, as it seems to cover most approaches to configuration modelling, it is a natural reference point for conceptual level analysis.

The remainder of this paper is organized as follows: An overview of software architecture and ADLs will be given in Section 2. Section 3 introduces our framework for analyzing and comparing ADLs along with the most important characteristics of three ADLs. In Section 4, a comparison between the ADLs and the concepts of the configuration ontology is presented. A mapping from the most important concepts of ADLs to the concepts of the configuration ontology is given in Section 5 and potential extensions of the ontology are discussed in Section 6. We discuss our findings and previous work in Section 7 and finally give our conclusions and topics for further research in Section 8.

2 SOFTWARE ARCHITECTURES AND ARCHITECTURE DESCRIPTION LANGUAGES

Software architecture of a system purports to describe the high-level structure of a software system. The significance of considering architecture when designing software systems is well understood. There is, however, no single, generally accepted method for describing software architecture. Simple methods, such as referring to an existing architectural style or using box-and-line diagrams with no or vague semantics, have been recognized to be inadequate for the task [13]. Hence, there is a need for better methods.

Architecture description languages (ADLs) are a promising candidate solution for the architecture description problem. Loosely defined, ADLs are formal notations with well-defined semantics, whose primary purpose is to represent the architecture of software systems. A large number of ADLs have been proposed. ADLs have in common the concept of component, although different ADLs have different names for the same concept [3]. But in their other characteristics, ADLs differ from each other radically. Some of them address a special application domain and others are dedicated to a specific architectural style [3]. ADLs also employ different

[1] Helsinki University of Technology, TAI Research Centre and Software Business and Engineering Institute, P.O. Box 9600, FIN-02015 HUT, Finland. Email: {Timo.Asikainen, Timo.Soininen, Tomi.Mannisto}@hut.fi

formalisms for specifying semantics, and there is variety in how rigorously the syntax and semantics are defined.

The most fundamental elements of architectural descriptions include *components*, *connectors* and their *configurations* [3,4,13].

Components represent the main computational elements and data stores of the system. Intuitively, they correspond to the boxes in the box-and-line diagrams. Clients, servers and filters are examples of components. In a working system, a component might manifest itself as an executable file or a dynamic link library. [4]

Unlike components, connectors are not loci of application specific computation in software systems. Instead, they represent interactions between components. In a box-and-line diagram, connectors are depicted as lines between the boxes. Examples of connectors include method invocation, pipes and event broadcast. [4]

Components can be connected to each other to form configurations. They are sometimes referred to as systems [4] or architectural configurations [3]. In many ADLs, components can only be connected through connectors; explicit use of connectors has even been proposed a defining characteristic of an ADL [3]. Typically, components are connected to each other through *connection points*. Different ADLs call these connection points with different names, e.g. port, role or interface.

In some ADLs, components can also have an inner structure. Such components are called *compound components* and they represent a subsystem that has an architecture of its own. With composite components it is important to be able to specify how the inner parts of the component are linked to the component itself. Usually, the linkage is defined by binding connection points of the compound component with connection points of its parts. Intuitively, binding means that the connection point of the compound component is in fact a connection point of some other component inside the compound component.

A practical concern with ADLs is the tool support available for them. Tool support is out of the scope of this paper, since the goal is to analyze the modelling languages. However, it should be noted that support for generating executable systems out of architectural descriptions is one of the goals of research on ADLs [3]. This is a goal shared by research on configuration modelling.

3 ANALYSIS OF THREE ARCHITECTURE DESCRIPTION LANGUAGES

In this section, we first define a framework for analyzing and comparing the concepts of ADLs with those of configuration. Thereafter, we use the framework to study three ADLs: Acme [4,5,6], Wright [7,8] and Koala [9,10].

3.1 Framework for analysis and comparison

The fundamental phenomena described by the configuration ontology and that presented in [12] are: taxonomies, structure, topology, resources, functions and constraints. Underlying all the above-mentioned phenomena is the division of configuration knowledge into three classes, *configuration model knowledge*, *configuration solution knowledge* and *requirements knowledge*. *Types* and *instances* are entities occurring in the configuration model knowledge and configuration solution knowledge, respectively.

In the following three subsections, we will analyse the above-mentioned ADLs using a comparison framework composed of three parts. The first part includes the key concepts of ADLs and the configuration ontology, and the relations between them. The concepts include *components*, *connectors*, *configurations*, *connection points*, *attributes*, *resources*, *functions* and *constraints*. The relations include *topology*, *taxonomy* and *structure*. The second part considers the existence of different concepts for types and instances. The last part of the framework is the variation mechanisms provided by ADLs and the configuration ontology.

3.2 Acme

The basic concepts of Acme are *components*, *connectors* and *systems*. System is the Acme term for configuration. On the other hand, there are no constructs for resources or functions in Acme. Both components and connectors have connection points that are called *ports* for components and *roles* for connectors. *Design elements* include component, connector, port and role. Components are connected to connectors by defining an *attachment* between the port of a component and the role of a connector. One connector may connect multiple components. Components cannot be connected directly to each other and neither can a connector to another connector. [4]

Components and connectors can have attributes that are called properties in Acme. Properties are uninterpreted values, i.e. they do not have any semantics defined.

In Acme, design constraints can be defined using first order predicate logic. They can be either *invariant* or *heuristic*: invariant constraints must hold, whereas heuristic constraints are merely hints of what should be true for an Acme system. Constraints can be used to express various aspects of Acme systems: e.g. the existence and values of properties and the connections present in a system. [5]

In addition, Acme includes a structure called *representations* that can be used for describing an alternative view of a component or a connector. *Rep-maps*, or in other words, *representation maps*, can be used to specify the correspondences between different representations of a design element. There is, however, no semantics defined for either representations or rep-maps. One possible use of these constructs is representing the compositional structure of a component and the correspondences between the ports and roles of the compound component and those of the contained components. [4].

Although types are not first class entities in Acme, it has two type systems: one for design element types and, and another for systems. Types in the design element type system are sets of required structure, i.e. design element declarations, and values. New types can be formed from existing types through subtyping. System types are called *families*. A family consists of design element type definitions. Subtypes of families can be formed through single or multiple inheritance. Also, a system can be declared to be a member of many family types. [6]

What makes types a secondary concept in Acme is that design elements and systems need not have a type or be a member of a family, respectively. A design element being of a given type merely implies that the design element has the structure and values specified by that type. Similarly with families, a system being a member of a family signals that the type definitions of the family are type definitions of the system, too. Therefore, type systems of Acme can be considered a sort of macro expansion mechanism.

The syntax and semantics of Acme are formally defined, the latter in terms of a mapping to first order predicate logic.

There seem to be no constructs in Acme for modelling variety. What seems to come closest to modelling variability is the family construct. It can be used to specify a set of type definitions shared by a set of systems. Furthermore, constraints can be used to enforce the instantiation of certain design elements. Hence, the family definitions complemented with constraints seem to provide a mechanism for specifying product families with certain properties.

3.3 Wright

As in Acme, there are *components*, *connectors*, *systems*, *ports*, *roles* and *attachments* in Wright and their semantics are the same in both languages. There are no attributes, resources or functions. What distinguishes Wright from Acme and makes it special among ADLs is its way of specifying the behaviour of ports, roles, connectors and components, and the possibilities for analysis based on these specifications. Wright uses CSP (Communication Sequential Processes) specified in [14], a formal approach for two purposes: (1) specifying processes that reside in Wright elements and (2) defining semantics of non-CSP parts of the language. In short, CSP is a formal method for specifying and analyzing the behaviour of objects in terms of sequences of events in which they engage. The pattern of events that is possible for an object is termed a *process*. [7,8]

Each port and role is associated with a CSP process. In addition, each connector and component includes a separate glue and computation process, respectively. The glue of a connector defines the operation of the connector as an entity. That is, the glue coordinates the operations of the other processes in the connector. Ports are attached to roles to form systems. Which ports can be attached to which roles, is determined by their process descriptions. The basic idea is that a port can be attached to a role if the port will behave well in all situations enabled by the role. In other words, CSP defines a compatibility relation between ports and roles.

The second usage of CSP in Wright, defining semantics of non-CSP parts of the language, allows using tools operating on CSP to reason about properties, most notably about dead-lock freedom, of a Wright connector. This is an important class of tool support enabled by the rigorously defined semantics of Wright.

Wright allows describing hierarchical structure of both components and connectors. This is done by enclosing a system into the place of a process. In addition to the normal system specification, bindings between the port and role names in the enclosing element and those specified in the enclosed system need to be specified.

Wright distinguishes between component and connector types and instances. Each connector and component is of exactly one type. There is, however, no taxonomy of types.

In addition to component and connector types, Wright includes a construct called *style*. Styles are collections of type definitions and *constraints*. They are expressed in first order predicate logic and they can be used in a manner similar to that in Acme described above. In addition to component and connector type definitions, a style can include *interface type* definitions. They are process descriptions that can be used in port and role definitions.

Type definitions in styles can be parameterized. That is, parts of the type definition can be left open and a value can be filled in when the type is instantiated. New styles can be defined in terms of existing ones through subtyping: the new style has the same type definitions and constraints as the old one plus some additional type definition or constraints.

Variation mechanisms of Wright are similarly limited as for Acme, although Acme uses the term family where Wright uses style. In short, styles supplemented with constraints seem to be able to express variability.

3.4 Koala

As the languages described above, the Koala model has *components* as a main design element. But in other respects, Koala differs greatly from its peers. In Koala, there is no notion of connectors, resources, functions or constraints. *Configurations* are comprised of components connected to each other through *interfaces* that are the connection points in Koala. The connection between components is not symmetric: a distinction is made between *provided* and *required interfaces*. Loosely defined, a component having a provided interface means that the component offers some service for other components to use. Similarly, a required interface signals a service being required by the component from some other component. Koala interfaces are similar to those in COM or Java. [10]

There are some limitations on how interfaces can be connected to each other: only required and provided interfaces of the same interface type can be connected with each other and each required interface must be connected to a single provided interface. On the other hand, a provided interface can be connected to any number of required interfaces, including zero.

In addition to connecting interfaces to each other, it is possible to connect constituent parts of interfaces directly. These parts are called *functions*. Hence, interfaces in Koala are not atomic even when considered as connection points.

Koala has a type system: a distinction is made between both interface and component types and instances. There is, however, no taxonomy of component or interface types.

Compound components can be used to express compositional structure in Koala, i.e. other components can be contained within a component. An interface of a compound component can be bound to an interface defined by a contained component.

Koala includes a construct, *module*, which is a component without an interface of its own. Modules are used inside compound components for gluing interfaces. Suppose, for example, that each component contained in a compound component has an initialization interface to be called before using the component. Due to binding rules, it would not be possible to bind all these interfaces to any single interface of the compound component. Therefore, a new configuration specific module is added: when the initialization function for the compound component is called, the call is routed to the module, which in turn calls the initialization functions of all necessary components in the order desired.

In addition to the constructs already mentioned, Koala provides mechanisms for handling both the *internal diversity* of components and the *structural diversity* in a configuration. Internal variety is manifested as variation of component parameters. There may be dependencies between parameters: a parameter value may imply that another parameter has a certain value. Structural diversity pertains to alternative provided interfaces for a required interface: e.g. there may be multiple components that provide the same interface required by a certain component. The choice between the interfaces is made by a construct called *switch* either statically, that is at compile time, if the information required for the selection is available, or, otherwise, dynamically at runtime.

We have no information about whether Koala has formally defined syntax or semantics.

4 COMPARISON OF CONCEPTS OF THE ADLS WITH THE CONFIGURATION ONTOLOGY

In this section, we use our framework defined in the previous section for comparing the concepts and constructs found in the ADLs with those of the configuration ontology.

4.1 Key Concepts and the Relations between Them

Component is the central concept of Acme, Wright and Koala. It is also present in the configuration ontology with that same name. The semantics are as well similar: components represent the defining parts of a system in configuration modelling, too. In addition, systems as defined in Acme and Wright and configurations as defined in Koala have a counterpart in the configuration ontology, namely configuration.

The notion of connection points is also common to all the studied modelling methods. In Acme and Wright they are called ports and roles in components and connectors, respectively. In Koala connection points are termed interfaces and in the ontology ports. The semantics of connection points are also similar in all the disciplines: they denote the mechanism for connecting other entities.

Connectors are first-class citizens in Acme and Wright. However, there are no connectors in the configuration ontology or Koala. Thus, there is a major difference in how the disciplines handle architectural connection – an important issue in both ADLs and in the configuration ontology.

What then is the reason for this disagreement in architectural connection? We believe that at least a partial reason for the importance of connectors in Acme and Wright can be found in the underlying assumptions of them and several of the ADLs not studied in this paper: a major issue in software architecture has been reusing existing components. Furthermore, there has been considerable effort in the software engineering community to reuse heterogeneous components, which cannot be connected directly to each other due to different communication mechanisms and various other reasons. Therefore, connectors have been introduced in ADLs as a vehicle for connecting heterogeneous components.

In Koala, the situation is rather different: components are homogenous and there seems to be no problem in connecting them directly, i.e. without connectors. Hence, Koala is much closer to the configuration ontology than Wright and Koala.

Resources, a feature present in the configuration ontology but not in any of the ADLs, is similar to the notion of provided and required interfaces present in Koala in the sense that they are both anti-symmetric. What is more, resources are produced and consumed by components, just as interfaces are provided and required. However, resources are produced and consumed in certain quantities, which gives them more expressive power compared with the notion of provided and required interfaces.

In addition to simulating provided and required interfaces, resources can be used to model other relevant quantities. Such quantities include memory, power, output capacity and throughput. The software engineering community has considered similar issues important [15]. Hence, resources could very well be an important feature of the configuration ontology when used to model software architecture.

Modelling functions is another feature of the configuration ontology that all three ADLs presented in this paper lack. Functions are an important aspect of software engineering usually termed features in the domain [16]. We believe that also functions could be very useful when modelling software with the ontology.

All the ADLs have some mechanisms for modelling structure. However, the configuration ontology provides much stronger mechanisms: the configuration ontology provides a wide range of variation mechanisms. Furthermore, in the configuration ontology a component can be a part of many components simultaneously, which is not possible in any of the ADLs.

All the disciplines except Koala have explicit mechanisms for expressing constraints. Further, in all disciplines where constraints exist, they are logical expressions about the non-behavioural properties of a system modelled in that discipline. A difference is that in the configuration ontology, there is no direct support for heuristic constraints as defined in Acme. Support for modelling preferences and optimization criteria have been identified as important and developed in other research on configuration.

4.2 Distinction between Types and Instances

All the three ADLs have some distinction between types and instances. In Acme, the distinction is rather weak, as the type systems can be seen as a simple macro expansion mechanism. Nevertheless, there is taxonomy between the Acme types. The situation is rather similar in Wright: types bear a little meaning as such. The only function of types seems to be facilitating in defining and altering recurring patterns. In Koala, interface types are strong in the sense that only interfaces of the same type can be connected. There is, however, no taxonomy for the interface types. The component types seem to have no function beyond defining the structure of a set of components. Hence, component types seem to be as a construct as weak as types in Acme and Wright.

In the configuration ontology, strong distinction between types and instances is one of the basic assumptions and is made for all kinds of entities. Types are organized in taxonomies.

4.3 Variation Mechanisms

A question closely related with the distinction of types and instances is: What is being modelled, one product or a product family. The configuration ontology aims at modelling product families. Configuration model knowledge defines the common properties of the family members. A lot of variation mechanisms are provided.

As stated in the analysis of Acme and Wright, both of these languages can be seen to provide some support for modelling variability: there are no explicit variation mechanisms, but the combination of system types and constraints seem to be able to express common structure shared by a set of products.

In Koala, there is some knowledge about the common properties of all the products: component and interfaces definitions are stored in a component repository and they are common to different systems to be constructed [10]. In fact, type definitions shared by a set of products is exactly the same phenomenon be have already seen in provided by the family construct Acme and by the style construct in Wright. As there are no constraints to complement the shared type definition in Koala, the support provided by Koala for variability is weaker than that Acme and Wright.

In the previous section, it was stated that Koala could model both internal and structural variety. How does this statement relate to the above observation that Koala provides a weaker support for variability than Acme and Wright? We claim that we are dealing with two distinct forms of variability. The variability in Acme and Wright can be used to span a set of products with many similarities, or in other words, a product family. On the other hand, the variation mechanisms in Koala seem to model behavioural variety of software embedded in a physical product instance: e.g. a television set can behave differently depending on some parameters. Of course, it could be argued that the television set in our example is, in fact, a product family. Nevertheless, we consider the variation mechanisms discussed above examples of different phenomena.

5 MODELLING SOFTWARE ARCHITECTURE WITH THE CONFIGURATION ONTOLOGY

In this section, we strive to synthesize the configuration ontology with the domain of software architecture. We do this by mapping the concepts in the ADLs to some concept or concepts in the configuration ontology. Components, ports, properties, and constraints are represented in the obvious manner using their direct counterparts, whereas the representation of connectors and roles is more problematic. Hence, we will present a mapping of connectors to components, and provided and required interfaces to ports with the aid of type specifications.

5.1 Modelling connectors as a type of component

In translating the semantics of connectors in Acme and Wright into concepts in the configuration ontology, it helps to observe that components and connectors have structures very similar to each other. Therefore, it is natural to view connector as a subtype of component with special semantic constraints. Indeed, defining connector to be a subtype of component will enable us to express part of the semantics associated with connectors. Furthermore, we can define roles in connectors to be ports in the connector-type components. To enforce the right use of connectors, we define suitable constraints that enforce the right use of connectors: e.g. in Wright, the only class of allowed connections is that between a component and a connector.

Subtyping can also be used for distinguishing provided and required interfaces from one other. By defining common supertypes for provided and required interfaces it is possible through multiple inheritance to have two versions of each port type, a provided and a required. By using constraints it is possible to assert that invariants concerning provided and required interface types hold. For instance, the fact that in Koala a required interface must be connected to exactly one provided interface of the same interface type can be easily captured using constraints.

5.2 Capturing diversity

Internal diversity of Koala can be captured with attributes defined by components and constraints. Dependencies between different parameters can be captured using constraints between attribute values of component types.

In the configuration ontology, cardinality of a port defines the amount of ports that can be connected to it. Cardinality can be used to capture some aspects of structural diversity in Koala. By defining cardinality greater than one for a port representing a required interface, multiple provided interfaces represented as ports could be connected to that port. This is only a partial solution as it says nothing about deciding which ports should actually be connected; constraints can be used to model this.

6 EXTENSIONS NEEDED FOR MODELLING SOFTWARE ARCHITECTURE

Albeit the configuration ontology captures a major part of aspects of all the studied ADLs, each of them has some features the modelling of which would require extending the ontology.

Capturing all of the idea behind **heuristic constraints** of Acme may require adding some method of representing optimization criteria and preferences in the configuration ontology.

There is no mechanism in the configuration ontology for **modelling behaviour** similar to the way how CSP is used in in Wright. In fact, the configuration ontology ignores behavioural aspects entirely. In case considering behaviour should be required in the configuration ontology, it would be natural to extend the constraint language to cover behavioural aspects, as the constraint language can be seen as the extension mechanism of the ontology.

Koala includes the method of **function binding**, in which the constituent functions of interfaces are connected directly to each other instead of connecting interfaces [10]. This construct gives an internal structure to Koala interfaces. Given that interfaces of Koala are modelled with ports in the configuration ontology, this contradicts with the underlying assumption of ports being undividable connection points. As a result, there is a mismatch between interfaces in Koala and ports in the configuration ontology.

There is a number of possible ways to capture ports with internal structure. The first one is to make Koala functions the basic level of connection. Unfortunately, this approach introduces major problems. Firstly, interfaces would lose their counterpart in the configuration ontology. Secondly, applying the approach would likely lead to increased complexity in models of software products: the fact that an interface can contain several functions implies this.

The second approach would be to introduce compositional structure for ports of the configuration ontology. Applied to the problem at hand, interface types correspond to port types that have ports corresponding to functions as their parts. This approach is appealing: it models the relation between interfaces and functions in a way corresponding to the intuitive understanding of the issue. This approach would require major changes to the ontology, however.

Binding of interfaces of a compound component with the interfaces of the inner parts is another feature of Koala lacking a counterpart in the ontology. It seems that the ontology would need to be extended in order for it to model this phenomenon.

7 DISCUSSION AND COMPARISON WITH PREVIOUS WORK

There is an apparent difference in the natures of the sets of product variations modelled in different disciplines. In the configuration domain, this set is typically termed as configurable product or a product family. One of the defining characteristics of this concept is a pre-designed general structure with a lot of variation in the configurations [17]. On the other hand, above it was found that Koala supports no common structure for a set of products. In fact, Koala is not targeted at modelling a product family or a set of them, but product populations, which are defined as a set of products with many commonalities but also with many differences [9]. Hence, the underlying aims of the ADL modeling and configuration modelling are not totally similar.

In the previous section it was stated that no satisfactory mapping could be found for function binding of Koala. One possibility to respond to this and similar problems is to ignore the problematic feature. Even though ignoring aspects of ADL that are of practical or theoretical importance is nothing we would do light-heartedly, we still believe that doing so in some cases will increase the usefulness of the configuration ontology in modelling software products. Therefore, the question is: which features of ADLs should be modelled. This is a question that can only be fully answered by empirical research into the nature of software product families.

Research closely related to this paper has been conducted earlier. We do not know of earlier attempts of comparing the concepts of software architecture description to those of configuration modelling. This is the main contribution of our paper.

In their work, Männistö et al. have pointed out the existence of the research area of configurable software and identified some key concerns in the area [18]. They have not, however, studied the concepts of ADLs in detail or proposed any mapping from these concepts to those of configuration modelling domain.

On the other hand, [19] presents a formalized software configuration management (SCM) ontology. The concepts of the SCM ontology are, however, different from those of the configuration ontology. They are aimed at representing the modules, files, or packages, their versions and the dependencies between these. The ontology does not take into account the connections and interfaces between components of a system.

Felfernig et al. have proposed a scheme for constructing configurators based on UML descriptions of configuration knowledge [12]. Basically, their approach could be used for creating configurators for software products as well. Their approach is, however, different from our approach: theirs is based on presenting configuration knowledge in UML, while our approach is based on modelling software with the concepts of product configuration.

In [20], Kühn has presented an approach to software configuration based on structure and behaviour. He uses statecharts, a method similar to finite state machines, for specifying the behaviour of a module. This approach is similar to Wright in that it describes both structure and behaviour. With its focus on using behavioural constraints for making decisions during the configuration process, this approach is different from ours.

8 CONCLUSIONS AND FUTURE WORK

Above, we have presented an analysis of three ADLs and compared their concepts to a conceptualization of configuration knowledge. The aim has been to find a mapping from the concepts of ADLs to those of the configuration ontology. Our goal is to use configuration ontology and its supporting toolset for configuring software.

We found counterparts and close correspondences in the configuration ontology for the main elements of the ADLs we have studied and were able to propose a mapping between them that shows that configuration languages can be used for representing architectural knowledge. For instance, both share the notion of components. Furthermore, compositional structure, systems formed of connected components and constraints are phenomena present in both disciplines. Hence, it seems that the concepts of the configuration ontology can be used for modelling software products. However, capturing some aspects of ADLs seems to require extending the configuration ontology. These aspects include function binding and binding the connection points of compound components with connection points in its inner parts. Another important aspect is modelling behaviour. Of the ADLs, Wright models behaviour. Additionally, the approach presented by Kühn also emphasizes behaviour [20]. The question whether behavioural aspects really are important and should be modelled when configuring software product families, should be resolved through empiric studies. The existence of Koala, a commercial ADL with no behaviour modelling, suggests that modelling behaviour is not absolutely necessary.

There are still open questions and a need for further work. It is necessary to define the mapping of the ADL concepts to the configuration ontology more rigorously. Moreover, an ontology and a configuration language for software products should be defined. This will probably require investigating more thoroughly the current ADLs and the conceptualizations of disciplines such as SCM, generative and feature based programming [16,21], and, of course, the developments in the UML community, as well as case studies of real software product families. After completing this, case studies are needed to verify the applicability of the configuration language to modelling software. Another issue to be concerned is the computational complexity of configuring software products. Theoretical complexity analysis can provide insight into this issue, but only experiments with real products will give relevant information on the practical feasibility from this point of view. When moving towards empirical studies, it is also necessary to consider which of the existing configurators and their modelling languages best support software configuration at a more detailed level than in this study.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support from Academy of Finland (project 51394) and National Technology Agency of Finland (Tekes).

REFERENCES

[1] J. Bosch, *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[2] S. Vestal, *A Cursory Overview and Comparison of Four Architecture Description Languages*. Technical Report. Honeywell Systems & Research Center, 1993

[3] N. Medvidovic and R. M. Taylor, 'A Classification and Comparison Framework for Software Architecture Description Languages', *IEEE Transactions on software engineering*, **26**, 70-93, (2000).

[4] D. Garlan, R. T. Monroe, and D. Wile, 'Acme: An Architecture Description Interchange Language', *in: Proceedings of CASCON'97*, 1997.

[5] D. Garlan, R. T. Monroe, and D. Wile, 'Acme: Architectural Description of Component-Based Systems', *in: Foundations of Component-Based Systems*, Cambridge University Press, 2000.

[6] R. T. Monroe, D. Garlan, and D. Wile. *Acme Reference Manual*. Available at "http://www-2.cs.cmu.edu/afs/cs/project/able/www/AcmeWeb/ACME%20StrawManual.html". Cited March 11, 2002

[7] R. Allen and D. Garlan, 'A Formal Basis for Architectural Connection', *ACM Transaction on Software Engineering*, (1997).

[8] R. Allen, A Formal Approach to Software Architecture. Doctoral dissertation, 1997.

[9] R. van Ommering, 'Configuration Management in Component Based Product Populations', *in: Proceedings of Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.

[10] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, 'The Koala Component Model for Consumer Electronics Software', *IEEE Computer*, **33**, 78-85, (2000).

[11] T. Soininen, J. Tiihonen, T.Männistö, and R.Sulonen, 'Towards a General Ontology of Configuration', *AI EDAM*, **12**, 357-372, (1998).

[12] A. Felfernig, G. Friedrich, and D. Jannach, 'UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems', *International Journal of Software Engineering and Knowledge Engineering*, **10**, 449-469, (2000).

[13] D. Garlan, 'Software Architecture', *in: Encyclopedia of Software Engineering*, J. J. Marciniak, ed. John Wiley & Sons, 2001.

[14] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[15] D. Garlan and D. E. Perry, 'Introduction to the Special Issue on Software Architecture', *IEEE Transactions on software engineering*, (1995).

[16] K. Czarnecki and U. W. Eisenecker, *Generative Programming*, Addison-Wesley, 2000.

[17] J. Tiihonen, T. Lehtonen, T. Soininen, et al, 'Modeling Configurable Product Families', *in: Proceedings of the 12th International Conference on Engineering Design (ICED'99)*, 1999.

[18] T. Männistö, T. Soininen, and R. Sulonen, 'Product Configuration View to Software Product Families', *in: Proceedings of Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.

[19] T. Syrjänen, 'Including Diagnostic Information in Configuration Models', *in: Proceedings of the First International Conference on Computational Logic*, 2000.

[20] K. Kühn, 'Modeling Structure and Behavior for Knowledge-Based Software Configuration', in: *Proceedings of the ECAI 2000 Workshop on New Results in Planning, Scheduling, and Design*, 2000.

[21] C. Prehofer, 'Feature-Oriented Programming: A Fresh Look at Objects', *in: Proceedings of ECOOP'97*, 1997.

Customizing the Interaction with the User in On-Line Configuration Systems

L. Ardissono[1] and **A. Felfernig**[2] and **G. Friedrich**[2] and **A. Goy**[1] and **D. Jannach**[2] and **M. Meyer**[3] and **G. Petrone**[1] and **R. Schäfer**[3] and **W. Schütz**[3] and **M. Zanker**[2]

Abstract. The provision of services on the Web is challenged by its heterogeneous customer base: Web catalogs are accessed by users differing in interests and knowledge about the products and services they search for. Moreover, in the companies selling complex configurable products and services, configuration systems are used by employees playing different roles: e.g., technical engineers and managers. For some of these people, the configuration task is problematic, as it exposes them to a large number of technical details to be specified. Effective personalisation strategies are thus critical to the development of successful Web-based configuration systems.

This paper presents the personalisation techniques applied in CAWICOMS, a prototype toolkit for the development of Web-based configuration systems that personalise the interaction with their users, supporting individual needs during the configuration task and the presentation of the solutions. The overall goal is that of assisting the user during the configuration process by suggesting suitable choices and providing her[4] with the information she needs for making informed decisions. To this purpose, our framework integrates user modelling and personalisation techniques with constraint-based configuration techniques.

1 INTRODUCTION

The provision of services on the Web is challenged by its heterogeneous customer base: as the popularity of Web shopping has dramatically increased, electronic catalogs are visited by users differing in interests and knowledge about the products and services they search for. The one-to-one recommendation of items, based on the recognition of the customer's preferences, has been introduced in several Web-based systems to help users find the goods best satisfying their needs [4, 5, 6, 11, 14, 17, 19]. However, this facility does not support the creation of personalised solutions on the fly: only pre-configured items can be managed by the traditional recommender systems. As the configuration of items is essential to comply with the customer's requirements when purchasing complex products, or registering for services, the need for assistance in this task is emerging as an important requirement. At the current stage, this type of activity can be performed by using non-personalised configuration systems offering a single type of interface. Moreover, as the configuration of complex items would challenge the user with technical details, the systems available on the Web typically solve simple configuration tasks, which can be reduced to exploring a pre-determined set of already configured solutions; for instance, see [8] and [9].

This paper presents the personalisation facilities offered by the CAWICOMS[5] framework for the personalised configuration of products and services [2, 3]. This framework is based on the idea that although, during the configuration of an item, technical details have to be addressed, an intelligent user interface can fill the gap between the system's point of view, focused on the implementation of the solutions, and the customer's one, focused on the usage of the product, or on the service fruition. This approach supports the use of the system by heterogeneous users such as inexperienced customers and technicians configuring items for third parties. CAWICOMS customises the following aspects of the interaction:

- *Configuration process:* during the selection of the features of the product/service, the system assists the user by suggesting, whenever possible, the values best fitting her requirements. Moreover, it provides information about such features, to help the user make informed decisions.
- *Presentation of solutions:* when a configuration solution is produced, the system presents it by focusing on the most interesting features, given the user's role. In this way, different perspectives on the product/service are offered (e.g., consider the presentation of technical details, with respect to the provision of information about the price and the main functions offered by the solution).

In order to achieve these two types of personalisation, the system applies user modelling strategies, aimed at identifying the user's individual interests and expertise, as well as the company's requirements. In several application domains, configuration systems are used by specific categories of users, identified by their roles in the organisations: e.g., sales engineers and managers. Our framework uses this type of information by applying stereotypical user modelling techniques [18], aimed at estimating the user's interests and domain expertise since the beginning of the interaction with the system. Moreover, to take into account individual characteristics, dynamic user modelling techniques are applied to update such estimates, according to the user's behaviour.

The system also employs probabilistic inference techniques to reason about the user's requirements and customise the configuration process. Furthermore, it uses personalisation strategies that, on the basis of the recognised interests, skills and requirements, prescribe the configuration and presentation actions to be performed: e.g., sug-

[1] Università di Torino, Italy.
[2] Computer Science and Manufacturing Research Group, University of Klagenfurt, Austria.
[3] DFKI, Saarbruecken, Germany.
[4] We refer to the user in a unique gender for readability purposes.

[5] CAWICOMS is the acronym for "**C**ustomer-**A**daptive **W**eb **I**nterface for the **C**onfi guration of Pr**o**ducts and Services with **M**ultiple **S**uppliers" (www.cawicoms.org). This work was funded by the EU through the IST Programme under contract IST-1999-10688.

gesting a value for a feature of the item to be configured, or focusing the presentation of a solution on a of its features. A rule-based approach is employed to tailor the interaction to the user's requirements and to integrate business rules in the configuration process.

The CAWICOMS framework has been applied to the development of a Web-based system supporting the configuration of high-technology products (telecommunication switches) and services (IP-VPN, Internet Protocol Virtual Private Networks) on the Web. In the present paper, we describe the user modeling and personalisation techniques applied in the system by referring to the prototype supporting the configuration of telecommunication switches.

This paper, which represents an extension of [1], is organised as follows: Section 2 outlines one of the application scenarios guiding the development of our framework and Section 3 summarises the main personalisation requirements on configuration we identified. Section 4 describes the CAWICOMS framework for the personalised configuration of items by specifying the typical flow of the interaction with the user (4.1), the graphical interface used to elicit information about the product/service features needed by the user (4.2), the knowledge representation for describing users and products/services (4.3), the personalisation strategies for customising the configuration task (4.4) and the inference techniques for reasoning about the user's interests and expertise (4.5). Section 5 sketches the system architecture and Section 6 concludes the presentation.

2 TELECOMMUNICATION SWITCHES DOMAIN

The development of the CAWICOMS framework was guided by application scenarios from the telecommunication domain. One of them is the configuration of large telecommunication switches for next-generation public telephony. The core of the product consists of a *switching network* that can be decomposed into a set of building blocks called *racks*, *frames* and *modules*. The number of these building blocks and their structure depends on the required performance characteristics and features specified by the customer. Therefore, the core of the switching system is complemented by routing components to conform future IP-based telephony requirements or software packages that offer control and maintenance functionality. In order to completely specify such a product, up to several hundreds of parameters and questions may be posed to the sales person interacting with the configuration system. Users can easily become overstrained and are unable to overview the configuration process. This is especially a problem when sales personnel is not well trained or lacks deep technical knowledge. In those cases, configuration systems play a crucial role as a corporate knowledge management tool, where user specific knowledge presentation requires an intelligent interface.

In our application domain we identified four user groups differing in the level of product knowledge and the frequency of system interaction: *Sales engineers* have deep technical knowledge. These users want to be able to drill down to configuration details and are able to interact with a non-personalised configuration system without any assistance. *Senior sales representatives* typically have good knowledge about the products and services to be configured and reasonable experience in the usage of configuration systems. *Junior sales representative*: this category encompasses sales personnel with almost no experience, and/or low level of technical understanding. *Customers*: the system is not allowed to assume any training on the product and must be prepared to give several explanations. This type of user is particularly important, when the configuration system is used as a medium to deliver product information.

In CAWICOMS, we addressed the requirements of these user classes, which are represented by stereotypes providing information about their skills and interests.

3 PERSONALISATION REQUIREMENTS OF CONFIGURATION TASKS

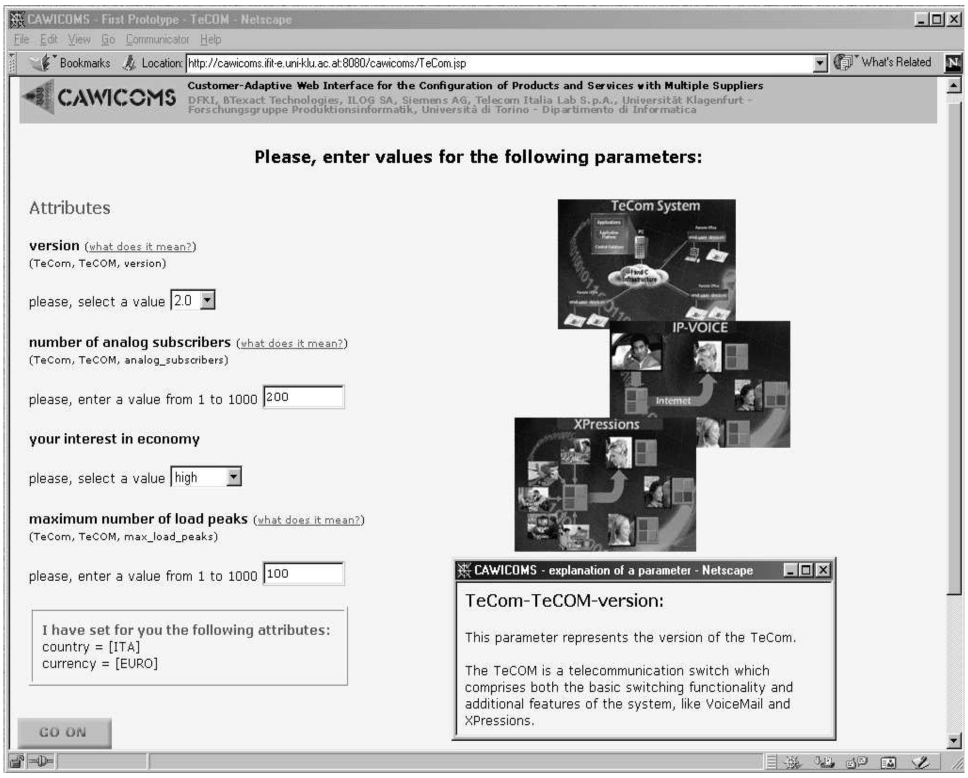
In order to take the user's interests and knowledge requirements into account, a configuration system should fill the gap between the underlying representation of the product/service and the user's perception of such an entity. While an expert user is assumed to have precise knowledge about the features of a product/service and its structure, a novice one only perceives its most "external" aspects. For instance, a telecommunication switch is characterised by a large set of features, some of which are very technical, such as the number of trunk lines to be exploited. However, the novice user's view of the product may only concern a subset of all the features: e.g., she may want to specify whether a Voice-mail function is needed, or how many terminals will be connected to the switch.

We have identified a set of requirements concerning the personalisation of the interaction by interviewing people regularly using the configuration systems available to a telecommunication company and occasional users of on-line configuration systems. The most relevant issues follow.

1. The configuration process may require the specification of a large set of data.
2. Depending on the user's knowledge, the specification of the parameter values may be difficult, if not impossible, as the user might not know the impact of her selections on the configuration solution.
3. Most users are only interested in the cost and the usage characteristics of the solution, while they do not care about how it is implemented (and therefore about the values to be set during the configuration process).
4. Some configuration parameters depend on the customer's features and should be automatically set: e.g., the customer's nationality determines the currency for payments.
5. Other configuration parameters are so critical that the user must take the responsibility to set them. At least in these cases, she should be supported with extra-helpful information about the parameter.
6. The user should be enabled to postpone some configuration decisions, when she is uncertain about the preferred value for a parameter, and carry on the configuration of the other aspects of the item under definition.

We have also identified requirements on the presentation of configuration solutions, but we only sketch them, as this paper is focused on the adaptation of the configuration task.

- The features of the configuration solution should be presented by taking the structure of the product/service into account. For instance, the structure of a telecommunication switch could be used to present the solution component by component, instead of showing a flat list of features (as most configuration systems do).
- Very critical information should be presented, regardless of the user's interests. However, if the information is too complex for the user, additional information should be available in order to help her understand the presentation.
- The presentation of solutions should be focused on the features most relevant to the user's interests: different types of information



CAWICOMS - First Prototype - TeCOM - Netscape

File Edit View Go Communicator Help

Bookmarks Location: http://cawicoms.ifit-e.uni-klu.ac.at:8080/cawicoms/TeCom.jsp What's Related

CAWICOMS Customer-Adaptive Web Interface for the Configuration of Products and Services with Multiple Suppliers
DFKI, BTexact Technologies, ILOG SA, Siemens AG, Telecom Italia Lab S.p.A., Universität Klagenfurt - Forschungsgruppe Produktionsinformatik, Università di Torino - Dipartimento di Informatica

Please, enter values for the following parameters:

Attributes

version (what does it mean?)
(TeCom, TeCOM, version)

please, select a value 2.0

number of analog subscribers (what does it mean?)
(TeCom, TeCOM, analog_subscribers)

please, enter a value from 1 to 1000 200

your interest in economy

please, select a value high

maximum number of load peaks (what does it mean?)
(TeCom, TeCOM, max_load_peaks)

please, enter a value from 1 to 1000 100

I have set for you the following attributes:
country = [ITA]
currency = [EURO]

GO ON

CAWICOMS - explanation of a parameter - Netscape

TeCom-TeCOM-version:

This parameter represents the version of the TeCom.

The TeCOM is a telecommunication switch which comprises both the basic switching functionality and additional features of the system, like VoiceMail and XPressions.

Figure 1. A personalised question page generated during a confi guration step.

about the configured product/service should be presented, depending on the user's role. For instance, a technical engineer should get technical information about the solution, while a manager could benefit from a presentation focused on its performance and economic aspects.

- The presentation should be tailored to the user's knowledge about the product/service: for instance, technical details should be hidden, if they are too complex for the user (and do not represent particularly critical information).
- As a general rule of Adaptive Hypermedia systems, the user must always be able to access the complete information about the solution. This means that details considered irrelevant, or too complex for the user, should be made available as additional information about the product/service. In this way, they can be reached on demand (by following "more info" links).

4 MANAGEMENT OF PERSONALISED CONFIGURATION TASKS

4.1 Interaction Flow

In the CAWICOMS system, the configuration process is organised in phases corresponding to the logical structure of the item to be configured: in each phase, a different component of the product/service is configured. Moreover, as each phase may consist of a possibly complex task, the interaction with the user within a phase is managed according to a dynamically generated sequence of configuration steps.

At each step, the user selects values for a subset of the product features: these features are represented by configuration parameters, each one associated with the set of its possible values (the domain of the parameter). After the selection of the parameter values, the user submits such values to the configuration engine. As in our framework this engine is based on a constraint satisfaction system, the values are propagated in a constraint network representing a partial solution. The propagation may trigger domain reductions on other parameters. After each propagation step, the user is shown another set of parameters to be set and their current domains, until the phase is over. Then, another configuration phase is started, until the whole product/service is specified. When the constraint network evolves to a solution where each parameter is set to one value, the system presents the solution. In contrast, if the user's choices generate a failure in the constraint propagation process, the configuration fails.

4.2 Management of a Configuration Step

The selection of parameter values within a configuration step is performed by filling in a form that shows the parameters to be set, together with their domain. The form may also include questions about the user's preferences for high-level properties of the product/service, such as its reliability. After having selected the values for the various questions, the user can submit the form to the configuration system by clicking on a "go on" button.

For instance, Figure 1 shows a typical page generated by our system during the configuration of a telecommunication switch (TeCOM). The leftmost part of the page displays the list of questions the user is asked about and includes configuration parameters (e.g., version of the switch, number of analog subscribers) and information about the customer's requirements: her interest in the economy of the product, i.e., on how costly the solution will be.

As shown in the figure, a help button ("what does it mean?" link) is available behind each parameter to retrieve detailed information

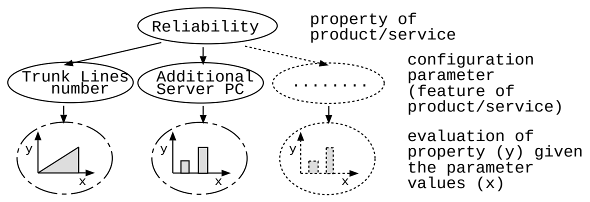

Reliability
property of product/service
Trunk Lines number
Additional Server PC
....
configuration parameter (feature of product/service)
y
x
evaluation of property (y) given the parameter values (x)

Figure 2. Relations between parameters and properties in the Frontend Model.

about its meaning: for instance, Figure 1 shows the explanation window for the "version" of the TeCOM. Notice also that, at each configuration step, the system may set some parameters, by applying personalisation techniques. When this happens, the system shows these settings below the list of questions. For instance, in Figure 1, the system has set country and currency for the switch.

4.3 Knowledge about Products/Services and Users

4.3.1 Introduction

The satisfaction of the requirements in Section 3 is based on the integration of user modelling, personalisation and flexible dialogue management techniques, and on the use of a domain ontology describing personalisation oriented information about products and services.

4.3.2 Representation of information about products and services

The technical knowledge about products and services is described in a *Product Model* supporting a conceptual, structured description of entities with features, components and constraints among components; see [10] and [7]. This model specifies the technical information needed by the configuration engine to generate solutions, but does not include high-level information typically addressed during the interaction with the user. This further type of information is stored in the *Frontend Model*, that extends the Product Model with data such as the explanation of the meaning of configuration parameters and an estimate of their technicality and of their criticality degrees. The Frontend Model also stores the impact of parameters on the utility of the solution regarding different aspects and the difficulty of knowing such information. For example, Figure 2 shows the representation of the impact of some product features (number of trunk lines, additional server PC) on the reliability of a switch. As shown in the figure, the number of trunk lines has positive impact on the reliability of a switch. Similarly, the number of additional servers enhances this product property.

4.3.3 Representation of information about users

The system manages an individual user model storing information about the user: this model is stored in the system's database, so that it is available after the first interaction, and includes various types of information:

- The user's *personal characteristics*, such as nationality and enterprise type, are represented as <feature-value> pairs;
- The user's *knowledge* about each product/service feature, corresponding to a configuration parameter, is represented as a probability distribution on the values of a binary variable, associated to the feature; this variable represents the system's estimates that the user knows/does not know the meaning of the feature.
- The user model also describes the user's *interests* in different aspects of the product, such as its reliability and economy, corresponding to the properties defined in the Frontend Model. These interests are represented as probability distributions on the values (levels) of variables associated to such properties. For each variable, three level of interest are considered: low, medium and high.
- The user model stores information about individual defaults, representing preferences for particular parameter values. This type of preference may be available because the system enables the user to set "long-lasting" preferences about product features.

The estimates about the user's interests and expertise are initialised by means of stereotypical information [18] about the most relevant classes of users interacting with the configuration system (private customers, sales representatives and technical engineers). Moreover, to take individual properties into account, the system's estimates are dynamically updated on the basis of an interpretation of the user's observable behaviour. (see Section 4.5).

4.4 Personalisation Strategies

The conceptual representation of products and services stored in the Frontend Model guides the system in the management of a structured configuration session, suggesting the configuration of the product/service one component after the other, in a possibly hierarchical order. However, the system enables the user to postpone the setting of parameters and to select the components that she wants to configure first. In this way, mixed-initiative dialogues are managed, where both the system and the user can take the initiative during the configuration process (requirement 6 in Section 3). The Frontend Model also supports the user during the setting of parameter values by providing her with with explanations of the meaning of the parameters to be filled in (requirement 5).

Finally, the assessment of the user's interests and expertise, together with the exploitation of the information stored in the Frontend Model, supports the satisfaction of the first four requirements, as it enables the system to automatically set parameters and to personalise the formulation of questions. Given a configuration parameter to be filled in, alternative strategies can be used to identify the value(s) to be set and a personalisation module evaluates the alternatives, searching for the most promising one:

1. *If the criticality of the parameter is over a threshold, then ask the user about the value to be set.*
2. *If the user model contains an individual default value for the parameter and the value is included in the current domain of the parameter, then set the parameter accordingly.*
3. *If a personalised default matching the user is available for the parameter and the intersection between the suggested values and the current domain of the parameter is not null, then set the parameter to the intersection.*
   Personalised defaults represent business rules suggesting parameter settings based on customer's characteristics and are represented as production rules. The head of the rule specifies a possibly complex and/or condition on the user data. The consequent suggests a set of values for the requested parameter, together with the result of the evaluation of the head on the user model. For instance, in the interaction of Figure 1, a simple personalised default is applied that sets the "currency" parameter of a telecommunication switch to the appropriate currency (USD vs. Euro) on the basis of the user's nationality.

4. *If the parameter is related to some properties for which the user's estimated interest is low, then set a standard (non personalised) default value consistent with the current domain.*
5. *If the user's estimated expertise is sufficient to choose a value for the parameter, then ask her to set the preferred value, given the current domain.*
   This strategy relies on a comparison between the user's expertise and the difficulty of the parameter in order to estimate the likelihood that the user will be able to answer the question [12].
6. *Given the parameter domain, select the best value and set it, given the user's interests in the product/service properties.*
   This strategy exploits the information in the user model to predict the preferred values for the parameter. The properties related to the parameter in the Frontend Model are used to focus on the corresponding user interests, which are analysed to check if a sufficiently substantiated prediction of the best value can be made. Section 4.5 describes the evaluation model ascribed to the user in our system.
7. *Elicit (if not yet done) information from the user about her interest in properties of the product/service that are influenced by the parameter to be set. Then, apply strategy 6 to possibly set the parameter values.*
   This strategy is applied to let the user self-assess her interests, when the information in the user model is not sufficient to perform any prediction.
8. *Postpone the parameter setting to a later stage of the configuration process (last resort).*

These strategies are sorted by priority because, whenever safe, automatic parameter settings are favoured over questions to the user. However, the selection of the strategy to be applied is a little more complex: while the evaluation of the first three strategies is binary (either they suit the current situation, or they do not), the other strategies rely on uncertain information. For instance, strategy 4 depends on the estimation of the user's interest for the properties related to the parameter in focus; similarly, strategy 5 is based on the probability that the user knows the meaning of the parameter. In order to take this uncertainty into account, the suitability of a strategy is evaluated, in the [0..1] range, and applicability thresholds are defined to rule out weak strategies.

For each parameter to be filled in, the personalisation module evaluates the strategies, according to their priority, and selects the first one exceeding the application threshold. The selected strategy is applied to continue the interaction with the user, either by eliciting information from her, or by autonomously setting the value. The question pages submitted to the user reflect the fact that the various parameters may be filled in according to alternative strategies. For instance, in Figure 1, the user is questioned about parameters and interests; moreover, some parameters are set by the system.

4.5 Reasoning About the User's Knowledge and Interests

For applying the personalisation strategies described in Section 4.4, the user's *interests* and the user's *expertise* have to be estimated based on her observable behaviour. The inference mechanism used for this estimation process has to take into account the uncertainty associated with the interpretation of the observations. This is the reason why we use a probabilistic inference mechanism, namely *Bayesian networks* [16], for this purpose.

For estimating the user's interests, we have to ascribe the user an evaluation process which she employs for assessing products and services. In an idealisation, we use Multi-Attribute Utility Theory (MAUT [21]) for this purpose. MAUT is a general evaluation scheme which is applied or at least compatible to the schemes applied by many user modelling approaches for estimating the user's interests [20]. Many users are also already familiar with MAUT, because it is used by consumer organisations for evaluating products. For example, in Germany, Stiftung Warentest uses MAUT for evaluating consumer products (e.g., digital cameras [22]). According to MAUT, the *overall evaluation* of an object determines its utility for the user. Usually, many aspects of an object can be evaluated and not all the users are interested in the same aspects to the same degree. These aspects are called *value dimensions*. For example, a telecommunication switch can be evaluated on the performance, reliability, and economy dimensions. In this example, some users are more interested in performance and reliability and less in economy.

The overall evaluation is expressed on a numerical scale, e.g., from 0 to 10 and the overall evaluation is defined as a weighted addition of the object's evaluation on its relevant value dimensions [21].[6] A weight is associated with each dimension to describe the user's interests. The more interested the user is, the bigger the weight is.

In the same way as for the overall evaluation, the evaluation of the object on each dimension is based on a weighted addition of the evaluation of the attributes relevant for this dimension. In order to evaluate attributes (in our configuration task, corresponding to parameters to be set), a numerical scale is constructed which represents the properties of the levels of an attribute (levels correspond to the parameter values). Then, an *evaluation function* maps evaluation values onto the attribute levels. For example, regarding reliability a guaranteed uptime of 99% yields 10, whereas an uptime of 50% yields 2. For simplicity, we assume that the evaluation functions of the attributes and their weights are fixed. The described weights and the evaluation functions are defined in the Frontend Model. See bottom of Figure 2 in Section 4.3.

For fulfilling the goal of estimating the user's interests, the weights associated with the dimensions have to be determined. In CAWICOMS, this is done by means of a probabilistic approach and the weights are represented as a probability distribution. Initially, a first estimate is obtained by using stereotypical knowledge about users. A set of stereotypes define categories of users, such as representatives of a small company. These stereotypes are activated based on the user's personal characteristics. Then, the user's observed behaviour in typical situations is interpreted in order to update these estimates. The following situations can be processed:

- Self assessment: especially at the beginning of the interaction, the system may ask the user about her interests. The user's answer reflects her self-assessment, which is very likely related to her interests, but this fact should not be taken for granted because the user might misunderstand the meaning of the terminology used by the system.
- The user can also change the parameter values that the system proposed as defaults by applying personalisation strategies. This type of action provides evidence that the user believes that the change has a positive impact on the overall evaluation of the configuration solution. In other words, the user believes that the new parameter settings cause a positive shift in the evaluation of the item to be configured, with respect to the evaluation with the values proposed by the system.
- After generating a configuration solution, the system presents it. Then, the user has to decide whether accepting the solution and

[6] Other possibilities for aggregation are described by [21].

ending the configuration process, or not. If she accepts the solution, her overall evaluation of the solution is probably quite good.

For each of these situations, a Bayesian network has been specified which reflects the above described dependencies. These specifications are domain independent. At runtime, the actual network for processing the situation with the parameters involved is created and used for the interpretation of the user's behaviour. This interpretation results in an update of the probability distributions representing the weights of the dimensions corresponding to the user's interests.

For estimating the user's expertise we use an approach based on [12]. For example, if a user is observed to click on a help button, she probably does not know the implications of the parameter and therefore her expertise is probably low. If we observe that the user knows the implications of a parameter (e.g., because she specifies a parameter value), her expertise is probably high.

5 SYSTEM ARCHITECTURE

The CAWICOMS system is based on a modular, distributed architecture, where a specialised module is associated with each main task to be carried out during the interaction with the user: e.g., configuration, user modelling, personalisation and generation of the Web pages. The system is implemented in Java and exploits standard software development environments. The user interface consist of a sequence of Web pages, implemented as JSPs, whose content is dynamically generated on the basis of the interaction context and the application of the personalisation strategies. The JSPs run within an Apache Web Server. Specialised, commercial engines are used within the system to perform complex tasks: for instance, the ILOG's JConfigurator engine [10, 15, 13] is used to generate configuration solutions.

6 DISCUSSION

We have presented the personalisation facilities offered by CAWICOMS, a framework for the Web-based configuration of products and services. These facilities allow tailoring the interaction style to the individual user and also support her in the configuration of the product/service which suits her needs best. The personalisation of the interaction is based on the integration of a user-oriented view of the configuration task with the technical level at which configuration systems usually work. This result is achieved by integrating constraint-based configuration techniques with user modelling, personalisation and dialogue management techniques. Our approach also uses a domain ontology describing personalisation-oriented information about the items to be configured. We have applied this framework to the development of a prototype system for the configuration of telecommunication switches; moreover, a second prototype, supporting the configuration of IP-VPNs, is under development.

We have performed a first test of the personalisation facilities offered by the telecommunication switches prototype, with a limited number of users having different background and playing different roles in their organisations (managers, technicians, etc.). The results show that such facilities are appreciated, especially as far as the automatic setting of parameters is concerned, because it speeds up the configuration process, leveraging the selection of the values for the parameters to be set. However, they want to control the system's decisions, possibly overriding them. For this reason, we have modified the user interface, to produce editable personalised suggestions: these suggestions should respect the user's preferences, but if they do not, she can correct the settings.

The users also appreciated the system's explanation capabilities, although only partially developed, because they shed light on the comples underlying configuration process.

Moreover, the users asked for more flexibility in the management of the dialogue between system and user. For instance, they suggested that the user should be enabled to notify the system that she does not care about a value, therefore the system should set it autonomously. We have incorporated these facilities in our prototype.

Finally, requests for a more flexible management of the overall configuration task came. For instance, the management of reconfiguration, with its implications (corrections of previous parameter settings, revision of a configuration solution, recovery from a configuration failure), was considered essential and is part of our future work.

REFERENCES

[1] L. Ardissono, A. Felfernig, G. Friedrich, A. Goy, D. Jannach, M. Meyer, G. Petrone, R. Schaefer, W. Schuetz, and M. Zanker. Personalising on-line confi guration of products and services. In *Proc. 15th Conf. ECAI*, to appear, Lyon, 2002.

[2] L. Ardissono, A. Felfernig, G. Friedrich, A. Goy, D. Jannach, R. Schaefer, and M. Zanker. Web-based commerce of complex products and services with multiple suppliers. In Thoben, editor, *E-Business Applications*, to appear. Springer Verlag, Berlin.

[3] L. Ardissono, A. Felfernig, G. Friedrich, D. Jannach, R. Schaefer, and M. Zanker. A framework for rapid development of advanced web-based confi gurator. In *Proc. 15th Conf. ECAI*, to appear, Lyon, 2002.

[4] L. Ardissono and A. Goy. Tailoring the interaction with users in Web stores. *User Modeling and User-Adapted Interaction*, 10(4):251–303, 2000.

[5] L. Ardissono, A. Goy, G. Petrone, and M. Segnan. Personalization in business-to-consumer interaction. *Communications of the ACM, Special Issue "The Adaptive Web"*, 45(5):52–53, 2002.

[6] BroadVision. Broadvision. http://www.broadvision.com.

[7] A. Felfernig, G. Friedrich, and D. Jannach. UML as domain specifi c language for the construction of knowledge-based confi guration systems. *Int. Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 10(4):449–469, 2000.

[8] FIAT. buy@fi at: la tua prossima auto. http://www.buy@fi at.com, 2001.

[9] Fidelity Investments. Insurance center @fi delity. http://www400.fi delity.com:80/, 2001.

[10] ILOG. ILOG JConfi gurator. http://www.ilog.com/products/jconfi gurator/, 2002.

[11] Net Perceptions Inc. Net perceptions. http://www.netperceptions.com.

[12] A. Jameson. *Knowing What Others Know: Studies in Intuitive Psychometrics*. PhD thesis, University of Amsterdam, 1990.

[13] U. Junker. Preference-programming for confi guration. In *IJCAI Confi guration Workshop*, pages 50–56, Seattle, 2001.

[14] A. Kobsa, J. Koenemann, and W. Pohl. Personalized hypermedia presentation techniques for improving online customer relationships. *The Knowledge Engineering Review*, 16(2):111–155, 2001.

[15] D. Mailharro. A classifi cation and constraint-based framework for confi guration. *AI in Engineering, Design and Manucturing*, 12:383–397, 1998.

[16] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[17] P. Resnick and H.R. Varian, editors. *Special Issue on Recommender Systems*, volume 40. Communications of the ACM, 1997.

[18] E. Rich. Stereotypes and user modeling. In A. Kobsa and W. Wahlster, editors, *User Models in Dialog Systems*, pages 35–51. Springer Verlag, Berlin, 1989.

[19] D. Riecken, editor. *Special Issue on Personalization*, volume 43. Communications of the ACM, 2000.

[20] R. Schäfer. Rules for using multi-attribute utility theory for estimating a user's interests. In *Proc. 9. GI-Workshops: ABIS-Adaptivität und Benutzermodellierung in interaktiven Softwaresystemen*, 2001.

[21] D. von Winterfeldt and W. Edwards. *Decision Analysis and Behavioral Research*. Cambridge University Press, Cambridge, UK, 1986.

[22] Stiftung Warentest. Digitalkameras: Pixeljagd. *test*, (6), 2000.

Deployment of Configurator in Industry : Towards a Load Estimation

Michel Aldanondo[1] - Guillaume Moynard[1,2]

Abstract: Our communication deals with configurator deployment in industry. Our goal is to define an estimation function that permit to quantify the workload necessary for deploying a configuration software in industry. As this work is in progress, we mainly focus on the configurator deployment problem and relevant estimation factor definitions. At the end of the paper, we provide some ideas about the estimation function.

I INTRODUCTION

Configuration software is becoming more and more frequently used by companies fighting in the mass-customization market. Almost all the ERP software proposes now configuration modules fully integrated in their offer as reported in [1]. The Lapeyre group, a European leader in industrial carpentry (windows, door, kitchen furniture…), has been being a significant user of configuration for almost 10 years. For the Lapeyre group and for many companies deploying configuration software, an important issue deals with the following questions (figure 1) : “for each configurator deployment, how many man-months of deployment capacity should I forecast ? what would be the resulting deployment cycle time ?”.

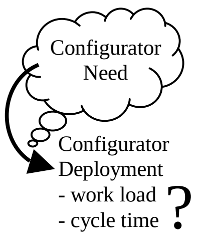


Figure 1 – The problem

If some scientific works have been achieved for this question for ERP deployment in terms of method and approaches, as far as we know, deployment load estimation for configuration software is not very frequent in the open scientific literature. Many configurator vendors have got their own method and approaches, but very few of them have been presented or discussed, some ideas can be found in [2]. Some rare estimation works in the field of product design are closer to our estimation goal, see [3] and [4].

Our goal is therefore to determine a quantitative estimation function for configurator deployment load ; the problem of the deployment cycle time will not be studied. This work, conducted by a Ph.D. student, is taking place in the Lapeyre group and gathers both selling aspect (product configuration) and manufacturing aspects (bill-of-materials configuration). This work is in progress and the presented results deal mainly with factors identification. Some ideas about the estimation function establishment will be provided.

The communication is divided in three parts. We first address the configuration deployment problem and underline the need for some estimation calculus. Then, we propose and discuss load estimation function factors. In the last part, we present the work that we intend to do concerning the load estimation function set up.

[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

2 THE CONFIGURATOR DEPLOYMENT LOAD ESTIMATION PROBLEM

2.1 Configurator deployment, a two step problem

Deploying a configurator can be decomposed in two steps : (i) modeling the product and (ii) coding the model in the configurator.

The purpose of the modeling step is “to put on the paper” the configurable product.
Normally, this activity should be conducted without taking into account the configurator that will be deployed. But in fact, most of the configurator vendors propose some modeling tools that fit their software. Some scientific modeling approaches can be found relying on various frameworks can be found in [5] for constraint satisfaction problem approaches or [6] for oriented object modeling.
As we target selling and manufacturing aspects, modeling is, most of the time, achieved by people from product marketing, product design and product manufacturing teams. The variety of persons and product knowledge involved in modeling makes this modeling phase very delicate. People from design and manufacturing “discover” what the marketing team thinks in terms of product offer and, at the opposite, marketing and selling persons “discover” that some new customization possibilities that they would never dare to ask are in fact very easy to handle. Sometimes getting these people around a modeling table generates conflict and withholding information is frequently observed. Thus, human factors and company organization are important factors for a smooth deployment achievement.
In our wok, modeling is decomposed in product configuration model design and bill-of-materials configuration model design. In order to discuss load deployment factors, we rely for (i) product model on a CSP based modeling approach (variable, domain, constraint) as shown explain in [5] and for (ii) bill-of-materials model on a generic bill of materials inspired form the work presented in [6]. Of course these two models are related in order to derive the configured bill-of-materials from the configured product.

The purpose of coding step is “to enter the product model” in the configurator.
In a perfect world, this activity should be conducted by somebody that knows only about configuration and configurator utilization without any product specific knowledge. As the product knowledge is gathered in the model on the “paper”, the person in charge of modeling should just translate the model into the coding entities existing in the configurator.
As it has already been pointed out, most of the time, modeling is a little bit dependant of the configurator. Therefore it happens sometimes, that model designing and coding are achieved by a same person. Nevertheless, we think that the load estimation function has to be considered for the two different steps, and we assume that the coding team members receive a valid model form the modeling team members. Once coding is done, an important sub-step is to check if the resulting configurator is valid. This is an important and hard issue (elements about this problem can be found in [7] or [8]) and such validation is most of the time done by the coding person with some help from model designers.

As a conclusion, the following aspects :

- many people involved from various company teams,
- organization of modeling and coding,
- information sharing problem,
- product and configurator knowledge distribution,

tend to generate long and difficult configurator deployment projects. Many companies measure configurator set-up workload in man-years of modeling, coding and maintenance. Thus, before deploying a configurator in a company or extending an existing configurator to a new facility or a new product family, being able to estimate the effort that will be done is of great interest. In order to do so, next sub-sections introduce the load estimation function.

2.2 Load estimation function set-up elements

The objective of the estimation function is to calculate the deployment work amount in load units "man-months" with respect to factors. Our goal is therefore to propose a function as :

Load = F ({factors})

In order to establish F, we will first identify factors characterizing :

- the industrial situation and organization in front of the deployment project,
- the product size and complexity.

Then, configuration deployment cases will be investigated in order to gather data provided by configurator deployment experts who are either project managers or consultants. The data can result :

- from effective deployment : quantitative values for factors and measured load,
- from quantitative estimation : factor values are provided to the expert who gives in return load estimation.

We will then have a set of cases ({factor value}, Load }, that should permit us to derive the estimation function.

2.3 Two levels of load estimation

One of the problems of project load estimation is to decide when and with what kind of information load estimation can be undertaken. Discussions with various configurator deployment experts allow concluding that two levels are interesting.

The first one corresponds with an order of magnitude estimation. This estimation must be conducted in around a single day of work involving a specialist of each company team (marketing, design, manufacturing and presumed project leader). The accuracy of the load estimation can be quantified around +/- 50%. Therefore this kind of load estimation is done before the configurator deployment final decision. We call this one : "rough estimation".

The second one is more detailed and can need up to ten days of work and requires some product modeling work. The idea is to estimate the load with some formalized elements concerning the configurable product. This allows reducing estimation inaccuracy down to +/- 20%. This kind of estimation is, most of the time, done once the configurator deployment is decided in order to plan the resouce deployment. We call this estimation : "detailed estimation".

2.4 Conclusions

The need for configurator deployment load estimation function has been shown. Two estimation levels, rough and detailed, have been introduced. Our estimation targets to cover selling purpose (product configuration) and production purpose (bill-of-materials configuration). Estimation factors are split in two groups : industrial situation factors and product factors.

Finally, an important point not already mentioned is relevant to the deployed configuration software. For this study, only one configuration software (provided by a configuration software company) is used, therefore the estimation function is valid for this single software. But each software has good and bad deployment capabilities and therefore we can assume that the results can be extended for other configurators.
The next section presents and discusses the estimation factors.

3 LOAD DEPLOYMENT FACTORS

The identified factors are the result of various configuration deployment case analysis. They must be considered as a first set that has been validated by configuration deployment experts. This set will be improved when the first estimation function will be tested.

3.1 Industrial situation estimation factors

These factors can be gathered in two sub-groups characterizing (i) the company know-how in terms of industrial organization and (ii) the organization of the persons involved in the deployment project. Once the factors of each sub-group are quantified, they can be aggregated in a single factor that will be used in the estimation function.

3.1.1 Company industrial organization factors

These factors characterize the general behavior of the company in front of any software deployment. These factors can be taken into account when estimating the load deployment of ERP or PDM systems for example. Four factors have been identified.

F1.1 : Management capabilities factor.
F1.1 characterizes :

- the existence of company projects dealing with : quality assurance, process reengineering, total quality control…
- the existence of regular utilization of project management techniques,

Factor quantification :

- if both exist : F1.1 = 1
- if none exists: F1.1 = -1
- if one of the two exists : F1.1 = 0

F1.2 : Sub-contracting factor
F1.2 characterizes if

- product manufacturing,
- analysis before software deployment,
- coding or customizing the software,

are subcontracted.
Factor quantification :

- if nothing is subcontracted : F1.2 = 1
- if one activity is subcontracted : F1.2 = 0
- if more than one activity is subcontracted : F1.2 = -1

F1.3 : Multi-lingual factor
F1.3 characterizes if the people involved in the deployment speak the same language and if the software will use more than one language.
Factor quantification :

- one language : F1.3 = 1
- more than one language : F1.3 = -1

F1.4 : Multi-location layout factor
F1.4 characterizes if sale, design and manufacturing are implemented in different physical locations.

Factor quantification :

- one location : F1.4 = 1
- more than one location : F1.4 = -1

The company industrial organization factor, F1, aggregates these four factors through a sum. Therefore FI ranges from –4 to +4 corresponding with very low and very good situation for software deployment.

3.1.2 : Configurator deployment involved person factors

These factors characterize the general organization and knowledge of the persons that could be involved in the configurator deployment. They are more specific to configurator deployment that the previous factors. Four factors have been identified.

F2.1 : Product knowledge existence factor
F2.1 characterizes if a single person with a good global knowledge level can be easily identified for each of the major following knowledge field :

- marketing and/or selling process,
- product design,
- product manufacturing.

Factor quantification :

- if the three knowledge fields are covered : F2.1 = 1
- if two of the knowledge fields are covered : F2.1 = 0
- if only one knowledge field is covered : F2.1 = -1

F2.2 : Modeling knowledge existence factor
F2.2 characterizes if a person in the company has already made some generic modeling, or if some models (whatever the formalism is) already exist for :

- product configuration, (for sale and design)
- bill-of-materials configuration. (for manufacturing)

Factor quantification :

- if both are covered : F2.2 = 1
- if one kind of model is covered : F2.2 = 0
- if nothing has been done in modeling : F2.2 = -1

F2.3 : Configuration model coding knowledge existence factor
F2.3 characterizes if a person in the company has already implemented in a configurator, a spreadsheet or in any computer language :

- product configuration model,
- bill-of-materials configuration model.

Factor quantification :

- if both have been coded : F2.3 = 1
- if one kind of model has been coded : F2.3 = 0
- if no coding has been done : F2.3 = -1

F2.4 : Project organization factor
F2.4 characterizes a possible distribution of knowledge and know-how of the people that could be involved in deployment. Product knowledge can deal with selling, designing and manufacturing, while know-how is relevant to modeling and coding of product and bill-of-materials. This can be summarized in the table of figure 2.

	Sale	Product design	Product manufacturing
Product Knowledge			
Modeling Know_how			
Coding Know_how			
	Product configuration		Bill-of-mats. configuration

Figure 2 – Skills and knowledge distribution

Factor quantification :
Figure 3, equivalent to figure 2 plus ellipses representing persons shows typical cases of knowledge and know-how distribution among the persons involved in the project.

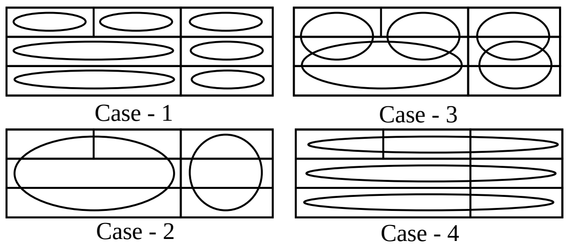


Figure 3 – Examples of Skills and knowledge distribution

Case 1 and 4 represents project organization with highly specialized person requiring strong and frequent information exchanges that will tend to slow the deployment.
Case 2 and 3 represents some kind of good complementary skill and know-how with some interesting overlapping in case 3.
Therefore, for organization close to :

- case 3 and 2 : F2.4 = 1,
- case 1 and 4 : F2.4 = -1.

As for factor F1, the configurator deployment involved person factor, F2, aggregates these four factors through a sum. F2 ranges from –4 to +4 corresponding with very low and very good project resource organization for configurator deployment.

3.1.3 : Conclusion on Industrial situation estimation factors.

Eight factors are used to characterize the industrial situation. These eight factors are aggregated in two factors ranging from [-4, +4] :

- F1 : Company industrial organization factor,
- F2 : Configurator deployment project involved person factor.

These two factors will be used to determine the estimation function. If gathered data coming from deployment cases is sufficient, it will be possible lately to consider the eight factors in the estimation function.
In terms of estimation level, rough and detailed estimation use exactly the same factors. Of course, for detailed estimation, as more time can be used to study the case, the industrial situation analysis provides much accurate value for these factors.

3.2 Product estimation factors

These factors target to take into account the product size and complexity in configurator deployment. By “size” we want to characterize the amount of characteristics needed for product and bill-of-materials configuration modeling and coding. Complexity tends to modulate this amount with the conceptual effort relevant to the interdependencies of characteristics.
We discuss rough estimation then detailed estimation. For each of them we will sequentially analyze product configuration and bill-of-materials configuration factors.

For these two estimations, the load estimation is conducted for a product family that has been identified as representative of all the company configurable products. Then this “product family load” is extrapolated to all products of the company. Therefore, families are identified with an extrapolation factor providing the family extrapolation factor set.

F3-4.1 : Family extrapolation factor
Factor quantification: F3-4.1: { (family_ref , extrapolation_factor) }
For example if two product families have been identified : family_ref_1 and family_ref_2. If :

- family_ref_1 has been chosen as representative
- F3-4.1 = { (family_ref_2, 0.6) },

Therefore : Load(family_ref_2) = 0.6 * Load(family_ref_1)

3.2.1 Product rough estimation factors

For this estimation, there is no modeling work just some rough analysis made through discussions with some people from marketing, design, and manufacturing.

3.2.1.2 Product configuration factors

As we rely on CSP modeling approaches for product configurable modeling, the factors will deal with variable number, variable definition domain size and constraints. The three following factors have been identified and must be quantified for the representative product family.

FR3.1 corresponds with an estimation of the number of configuration variables. We mean by configuration variables, variables that would be present in a CSP based model.
Factor quantification : FR3.1 : number of variables

FR3.2 corresponds with an estimation of an average variable value number.
Factor quantification : FR3.2 : average number of variable values

FR3.3 corresponds with an estimation of complexity. For us complexity means the degree of interdependence existing between the variables. In a CSP based model, this could represent some kind of ratio taking into account : variable quantity and number of constraints (gathering compatibility and activity constraints in the sense of DCSP approach of Mittal et al [9]). As no modeling is done for the rough estimation, complexity is estimated with the selection in figure 4 of one of the four cases representing different level of variable interdependence. Each small cross represents a configuration variable and each line a constraint between the two variables.

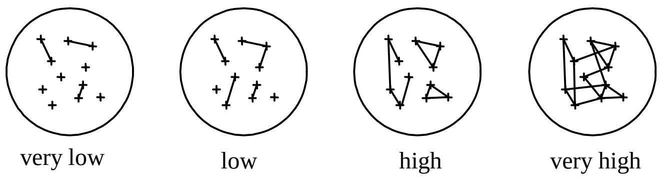


Figure 4 – Product complexity estimation

Factor quantification :
- very low complexity : FR3.3 = -2
- low complexity : FR3.3 = -1
- high complexity : FR3.3 = 1
- very high complexity : FR3.3 = 2

With the proposed factors, it is to be noticed that identification of the number of product families is of importance and has an important impact on the estimation.
Let us consider, for example, a case where the configurable products are windows and doors. It is possible to consider :
- either 2 families with for each family :
  - number of variables, FR3.1 = 15
  - complexity : low
- or one family
  - number of variables, FR3.1 = 20
  - complexity : very high.

It is not easy to decide what is the best solution. The two family case :
- limits modeling and coding complexity during deployment,
- increases the amount of basic modeling and coding due to the total number of variables
- increases maintenance effort by having two models and relevant pieces of code to maintain instead of one.

Interesting elements concerning this hard problem of family identification can be found in [10].

3.2.1.2 Bill-of-materials configuration factors

As bill-of-materials generic modeling consist of a hierarchical physical decomposition of the product. The resulting model shows :
- standard components (with frozen characteristics) at the lowest level of the bill-of-materials,
- generic sub-assemblies for the intermediate levels,
- father-child links with some conditional existence rule between :
  - upper level configured product and generic sub-assemblies,
  - generic sub-assemblies of different levels,
  - generic sub-assembly and standard components.

For rough estimation, factors just deal with standard components and bill-or-materials decomposition levels.

FR4.1 corresponds with an estimation of the number of different standard components that can be present at the lowest level of the generic model bill-of-materials.
Factor quantification : FR4.1 : number of standard components

FR4.2 corresponds with an estimation of the number of decomposition levels that should be present in the generic model of the hierarchical bill-of-materials of the family.
Factor quantification : FR4.2 : number of bill-of-materials level

These two factors permit in fact to get an order of magnitude of the number of generic sub-assemblies and relevant bill-of-materials links, which correspond to the modeling and coding workload.
As constraints are mainly taken into account in the CSP based product model, we do not take into account a complexity factor for generic bill-of-materials modeling and coding.

3.2.1.3 Conclusion about product rough estimation factors

The proposed factors have been discussed with different people who are familiar with configurator deployment problems. We think that three points might be subject to discussion and may need further investigation :
- FR3.2 : when the configuration variable definition domain is continuous (for example when dealing with parametric or tailored components) the meaning of “average number of variable values” should be different.
- FR3.3 : the discretization of product complexity in four levels,
- FR4 factors : the fact that bill-of-material complexity is not taken into account.

3.2.2 Product detailed estimation factors

When this estimation is processed, the generic modeling of the representative family has been done for both product and bill-of – materials configuration. Therefore, the estimations of some factors are replaced by measurement achieved on the models.

3.2.2.2 Product configuration factors

As the CSP based model is on the “paper”, the factors F3.1 and F3.2 are the same except that the numbers are counted on the model. For F3.3, the estimated complexity is replaced by the number of constraints that are present in the model.

Factor quantification : FD3.1 : number of variables
Factor quantification : FD3.2 : average number of variable values
Factor quantification : FD3.3 : number of constraints

At this time of the study, in order to be consistent with the complexity factor FR3.3, we consider product complexity close to the ratio : number of variables / number of constraints.

3.2.2.2 Bill-of-materials configuration factors

As in the previous sub-section factor values are not estimation anymore but rely on the bill-of-materials model analysis. The definition of FD4.1 is unchanged. But for FD4.2, instead of the number of hierarchical levels of the generic bill-of-materials, we consider the number of generic sub-assemblies that are present in the model.

Factor quantification : FD4.1 : number of standard components
Factor quantification : FD4.2 : number of generic sub-assemblies

At this time of the study, we do not take into account the number of generic links of the bill-of-materials and the number of constraints or rules modulating links existence. This might be necessary in a very close future.

3.2.2.3 Conclusion about product detailed estimation factors

As for the rough estimation factors, these factors have been discussed. The main improvement that have been proposed is to take into account the fact that pieces of product and pieces of bill-of-materials (or sub-assemblies) might be used several times for a same product family.
For product modeling and coding, this means that some configuration variables grouping is possible. For example, a configurable fastening device may be configured more than one time during the configuration of a single window. In order to avoid multiple modeling and coding of a same device many configurators and some modeling approaches [11] or [12] propose a notion of “group of configuration variables” with re-use possibilities.
For bill-of-materials modeling and coding, it is obvious that a single generic sub-assembly can be re-used in different branches of the generic hierarchical bill-of-materials. For example, considering the configurable fastening device as a generic sub-assembly, it is clear that this sub-assembly will be modeled one time and the corresponding piece of code re-used in the configurator when necessary.

3.2.3 Conclusions about product estimation factors

The proposed set of factors must be considered like a first proposition. We are, at that time, far from the final validation, but we think that the proposed elements permit to establish a good first approach of the estimation problem.
For the family load extrapolation mechanism introduced in the beginning of section 3.2 that permits to determine, from the analysis of a single product family, the deployment load of all product families, it is obvious that the re-use problem presented in section 3.2.2.3 is to be considered. This must provide a re-use factor between families for both modeling and coding of both product and bill-of-materials.

F3-4.2 : Family re-use factor
Factor quantification: F3-4.2:
{ (family_ref_x, family_ref_y, re-use_factor) }
For example if two product families have been identified : family_ref_1 and family_ref_2. If :
- family_ref_1 has been chosen as representative,
- F3-4.1 = { (family_ref_2, 0.6) },
- F3-4.2 = { (family_ref_1, family_ref_2, 0.1) },

Therefore : Load(family_ref_2) = (1-0.1)*0.6*Load(family_ref_1)

4 LOAD ESTIMATION FUNCTION

At the present time, we are working on the determination of the load estimation function and this section presents our current ideas about :
- the estimation function shape we hope to get,
- the deployment load data collection,
- cases gathering.

4.1 Estimation function

The estimated function will be characterize by the following elements :
- The estimated deployment load for product configuration (Load_3) and bill-of-materials configuration (Load_4) are additive. This mean that two independent load estimation function, f_3 and f_4, respectively for product and bill-of-materials should be identified :
  - Load_3 = f_3(F3.1, F3.2, F3.3)
  - Load_4 = f_4(F4.1, F4.2)
- These two deployment loads are added and modulated, with a function f_1, according to industrial situation estimation factors introduced in section 3.1. Company industrial organization factors (section 3.1.1) and configurator deployment project involved person factors (section 3.1.2) are first aggregated in a function f_2.

These assumptions permit to present the shape of the estimation function we target :
F ({factors}) =
f_1 [f_2 (F1, F2) , f_3(F3.1, F3.2, F3.3) + f_4(F4.1, F4.2)]

The major drawback of the estimation function shape is that industrial situation estimation factors modulate in a same way product and bill-of-materials deployment load. This could be of course decomposed, but the number of investigated cases would be necessary much larger.

4.2 Deployment load data collection

As explain in section 2.2, data can result from effective deployment measurements or quantitative estimation of theoretical deployment cases.
For both data sources, once factor values are identified the deployment load is gathered in a table as shown in figure 5. This table allows collecting data in various ways :
- detailed data collection, with load corresponding with (1) combinations, that allows the most detailed statistical analysis,
- semi-detailed data collection, with load corresponding with (2) or (3) combination allowing some differentiation according to product/bill-of-materials or modeling/coding in the statistical analysis,
- global data collection, corresponding with the total deployment load (4) that does not permit any differentiation.

	Product	Bill-of-material	
Modeling	(1)	(1)	(3)
Coding	(1)	(1)	(3)
	(2)	(2)	(4)

Figure 5 – Data collection possibilities

This way to collect data is interesting mainly for completed effective deployment, where project decomposition can vary and load measurements can not provide detailed data. Most of the time project decomposition is done according to the product/bill-of-materials deployment criterion and relevant set of data (2) is most often collected.

4.3 Cases gathering.

We try to gather cases according to common experimental design rules that permit to avoid that some factor level disturb the estimation function while taking into account some factors correlation possible existences.

5 CONCLUSION

We have presented in this communication elements trying to calculate a load estimation function for configurator deployment.
This work is in progress and we are currently gathering data. We hope to be able very soon to show some quantitative results in terms of functions.
The proposed set of factors need a strong validation and some improvements will be necessary. Nevertheless, they can be considered as a first basis for discussion.

During this factor identification work, all the discussions we have been having in industry with people involved in configuration software deployment, permit to conclude that load deployment estimation is a main issue for companies that either provide or use configuration software.

REFERENCES

[1] Gartner Group. Supercharge the Sales Effort*: Sales Application for Large Enterprise*, Strategic Report Analysis n° R-14-4280, Gartner Research, September 2001.

[2] 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.

[3] H.A. Bashir and V. Thomson, *A quantitative estimation methodology for design project.* IEPM 1999 conference, Vol 2 pp 198-506, Glasgow UK, july 1999.

[4] A.T. Bahill and W.L. Chapman. *Case studies in system design*. Workshop on system engineering and computer based system. pp 43-50, Picataway NewJersey USA 1995.

[5] M. Aldanondo M., G. Moynard and K. Hadj-Hamou., *General configurator requirements and modeling elements,* ECAI Workshop on Configuration, Berlin, Germany, pp. 1-6, 2000.

[6] A. Felfering, G. Friedrich and D. Jannach, *UML as domain specific language for the construction of knowledge base configuration system,* 11th Int Software Engineering and Knowledge Engineering conference, Kaserlautern, Germany, pp 337-345, 1999.

[7] A. Felfering, G.Friedrich, J. Dietmar and M. Stumptner, *Exploiting structural abstraction for consistency based diagnosis of large configurator knowledge bases*, ECAI Workshop on Configuration, Berlin, Germany, pp 23-28, 2000.

[8] M. Sabin, M. and E.C. Freuder, *Detecting and resolving inconsistenciy and redundancy in conditional constraint satisfaction problems*, AAAI Workshop on Configuration, Orlando, USA, pp 90-94, 1999.

[9] S. Mittal and B. Falkenhainer, *Dynamic Constraint Satisfaction Problems ,* 9th National Conference on Artificial Intelligence AAAI, Boston, USA, pp 25-32, 1990.

[10] N. H. Mortensen, B. Yu, H.S. Skovgaard and U. Harlou, *Conceputal modeling of product families in configuration projects,* ECAI Workshop on Configuration, Berlin, Germany, pp. 68-73, 2000.

[11] D. Sabin and E.C. Freuder, *Configuration as Composite Constraint Satisfaction*, Proceedings of the Artificial Intelligent and Manufacturing Research Planning Workshop, AAAI Press, pp. 153-16, 1996.

[12] M. Véron and M. Aldanondo, *Yet another approach to CCSP for configuration problem*, ECAI Workshop on Configuration, Berlin Germany, pp 59-62, 2000.