INTERNATIONAL CONFERENCE ON ENGINEERING DESIGN
ICED 03 STOCKHOLM, AUGUST 19-21, 2003

A PRACTICAL TOOL FOR MASS-CUSTOMISING CONFIGURABLE PRODUCTS

Juha Tiihonen, Timo Soininen, Ilkka Niemelä, and Reijo Sulonen

Abstract

Configurable products are an important way to achieve mass-customisation to satisfy individual customer requirements. We describe a novel configurator prototype that supports tailoring, i.e. configuring, such a product. It contains a semi-visual modelling tool based on a high-level object- and product structure-oriented modelling language with a clear formal semantics. The main functionality is delivered by a configuration support tool aimed at e-commerce. This tool provides intelligent support for configuring a product by applying a state-of-the-art inference engine for the form of logic programs on which the formal semantics is based. The user interface of the tool is generated almost automatically. We have modelled four real products from two domains and tested the efficiency of the inference engine on them. For this, we use a modelling-language-independent method for systematic testing. For these products, the modelling language is suitable and the inference engine efficient enough for practical use.

Keywords: configuration modelling, customisation, product families, product modelling, web-based systems.

1 Introduction

Mass-customisation aims at satisfying the specific needs of individual customers with mass-production-like efficiency. One important way to achieve mass-customisation is to develop and market *configurable products* [1]. A configurable product is designed once and this advance design is used repetitively in the sales-delivery process to produce a specification of a *product individual* that meets the customer requirements. This repetitive *configuration task* requires only limited and systemised variant design, which makes the sales-delivery process simple and the lead-time short. All the information on the possibilities of adapting the product to customer needs is described in a *configuration model* that defines the set of pre-designed components, and rules on how these can be combined into valid product individuals.

A *product configurator* (or *configurator* for short) is an information system that enables the creation and management of configuration models and supports the configuration task. A configurator generates a specification of a product individual that meets the given customer requirements and complies with the configuration model. This support is based on applying artificial intelligence techniques that provide inference on the basis of a configuration model and requirements specified by the user. Several formal models of configuration knowledge and tasks based on e.g. constraint satisfaction problems (CSP), rule-based reasoning, and different logical formalisms have been proposed and implemented [1], [2]. Each has its strengths and weaknesses and there is no dominant approach. Many of them have also been shown potentially computationally very expensive [3], which could make them impractical. There are some documented results on the practical efficiency of configurators, e.g. [4], [5], [6], but thorough and wide range empirical testing of configurators on real products is still lacking.

In this paper we give requirements for a practical configurator intended to support e-commerce (Section 2). Then we describe the core features, architecture and key design solutions of a web-based configurator prototype WeCoTin (acronym for Web Configuration Technology) that corresponds to needs of companies selling configurable products (Section 3). In addition, we describe a test method for empirical performance testing of configurators. The method is based on the idea of simulating a naïve user inputting random requirements to a configurator. We give results of performance testing on one of the four products modelled to validate WeCoTin (Section 4). Finally, we discuss and compare WeCoTin and our results with related work, summarise our experiences on the configuration modelling method (Section 5), and present conclusions and topics for further work (Section 6).

2 Practical requirements for a configurator

In this section we present central requirements specific for a practical web-based configurator. The requirements were identified in joint projects with manufacturing industry and our previous work, e.g. [7]. In the following, *modeller* refers to a person who creates and maintains configuration models and related information, and an *end-user* (or *user*) configures a product.

Configuration models need to be changed to reflect the changes in product offering. Long-term management of configuration models has often been a problem; an extreme example, the R1/XCON system, is documented in [8]. To facilitate long-term management, product experts such as product managers should be able to model the products. This avoids the cost of experts such as knowledge engineers or programmers that are traditionally needed to maintain configurators and eliminates the error-prone communicating of product knowledge to separate modellers who are not product experts. Modelling should be easy-to-understand for product experts, and *declarative* allowing the modeller to specify *what* kind of product individuals are valid instead of procedural requiring specification of *how* to create them. The modelling language should be object-oriented to divide configuration models into relatively independent pieces with low complexity and to exploit their common characteristics. It should be straightforward to model typical configuration phenomena such as alternative components in a product structure. The user interface for end-users should require little work and no programming to create and maintain when products change. In addition to effortless modelling, advanced long-term management requires support for modelling the evolution of products, components and their interdependencies in a way resembling configuration management (CM) and product data management (PDM). Further, it should be possible to efficiently deploy configuration models to sales people and customers without the risk of using out-dated configuration models, and multiple users should be able to configure products simultaneously. Configurations should be exportable to e-commerce, ERP, or PDM systems, etc. for further order processing.

Fundamentally, a configurator must check a configuration for *completeness* (i.e. that all the necessary selections are made) and *consistency* (i.e. that no rules are violated) with respect to the configuration model. It should be impossible to order an inconsistent or incomplete configuration. The user should be further supported by deducing fully the consequences of previous selections. This means, e.g., automatically making selections implied by the previous selections, identifying alternatives incompatible with them, and ensuring at each stage of the configuration task that the user does not end-up in a "dead-end" that cannot be completed into a complete configuration due to previous selections. In addition, explanations for incompatibility of selections should be available. This helps users in learning the product and its restrictions. However, it should be possible to make incompatible selections, which can help an expert user to quickly modify the configuration. Ease and flexibility of use for non-expert users of a web-based configurator implies a number of specific requirements. The user should be kept aware of selections that have been made and that must still be made, and about state of completeness (complete, incomplete) and consistency (consistent, inconsistent) of the configuration. It should be possible to guide a non-expert user through selections, but allow experts to make selections in different order. Further, the configurator should be accessible to any customer who can use a web-browser, preferably in his own language.

3 WeCoTin Configurator

In this section we describe WeCoTin, a configurator prototype whose high-level architecture and main functionality reflect the requirements set above. WeCoTin consists of two main components: a graphical modelling environment *Modelling Tool* for modellers (Section 3.1, shown on the right in Figure 1) and the web-based *WeCoTin Configuration Tool* that supports the configuration task (Section 3.2, shown on the left in Figure 1). WeCoTin is implemented with the Java 2 Platform and Java programming language, except when noted.

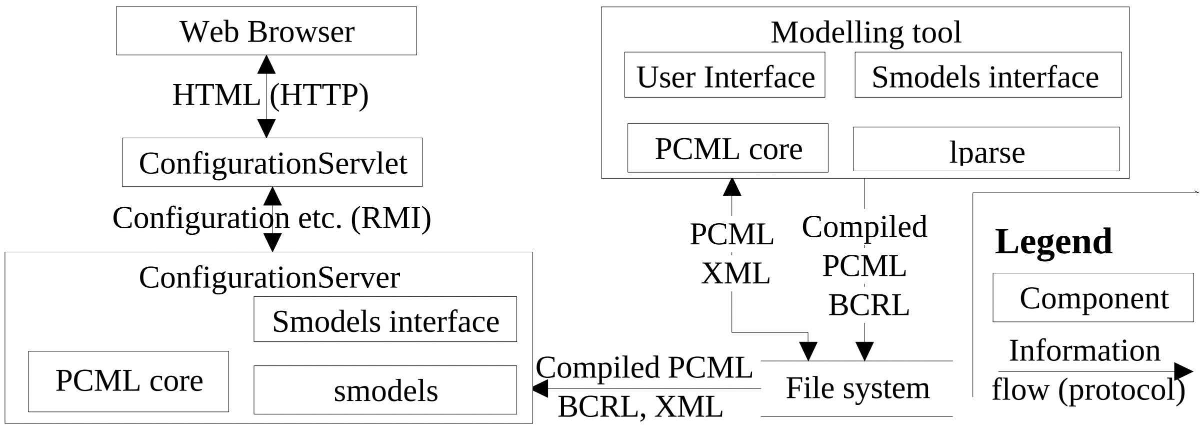
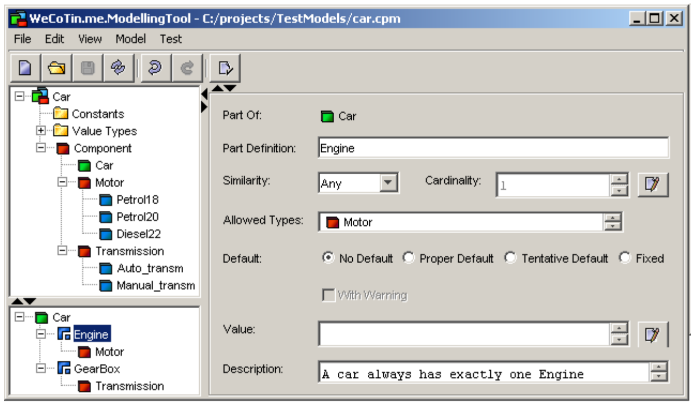


Figure 1. WeCoTin Architecture: Configuration Tool on the left and Modelling Tool on the right

3.1 WeCoTin Modelling Tool

Modelling Tool is used for creating and editing configuration models and information needed to generate a user interface for end-users. Configuration models are expressed in Product Configuration Modelling Language, *PCML*. The component “User Interface” in Modelling Tool (Figure 1) displays the current configuration model and facilitates editing it. It modifies the data structures representing the configuration model in component “PCML core” (Figure 1) according to modeller actions. A save operation stores the configuration model as PCML. The tool also compiles the configuration models for use in WeCoTin Configuration Tool.

PCML is object-oriented and based on a practically important subset of a synthesised ontology of configuration knowledge [9]. The main concepts of PCML are *component types*, their *compositional structure*, *properties* of components, and *constraints*. Component types define the parts and properties of their *individuals* that can appear in a configuration. A component type defines its compositional structure through a set of *part definitions*. A part definition specifies a *part name*, a non-empty set of *possible part types* (*allowed types* for brevity) and a *cardinality* indicating the possible number of parts. A component type may define properties that parametrise or otherwise characterise the type. A *property definition* consists of a *property name*, a *property value type* and *a necessity definition* indicating if the property must be given a value in a complete configuration. Component types are organised in a *class hierarchy* where a *subtype* inherits the property and part definitions of its *supertypes* in the usual manner. A component type is either *abstract* or *concrete*. Only an individual directly of a concrete type can be used in a configuration. Constraints associated with component types define conditions that a correct configuration must satisfy. A constraint expression is constructed from references to parts and properties of components and constants such as integers. These can be combined into complex expressions using relational operators and Boolean connectives.



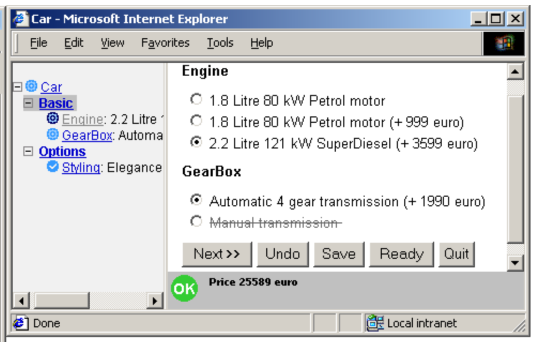
WeCoTin.me.ModellingTool - C:/projects/TestModels/car.cpm

File Edit View Model Test

- Car
  - Constants
  - Value Types
  - Component
    - Car
    - Motor
      - Petrol18
      - Petrol20
      - Diesel22
    - Transmission
      - Auto_transm
      - Manual_transm

- Car
  - Engine
    - Motor
  - GearBox
    - Transmission

Part Of:	Car
Part Definition:	Engine
Similarity:	Any
Cardinality:	1
Allowed Types:	Motor
Default:	[✓ No Default] Proper Default Tentative Default Fixed
	With Warning
Value:	
Description:	A car always has exactly one Engine



Car - Microsoft Internet Explorer

File Edit View Favorites Tools Help

- Car
  - Basic
    - Engine: 2.2 Litre
    - GearBox: Automa
  - Options
    - Styling: Elegance

Engine

- 1.8 Litre 80 kW Petrol motor
- 1.8 Litre 80 kW Petrol motor (+ 999 euro)
- [✓ 2.2 Litre 121 kW SuperDiesel (+ 3599 euro)]

GearBox

- [✓ Automatic 4 gear transmission (+ 1990 euro)]
- ~~Manual transmission~~

Next >> Undo Save Ready Quit

OK **Price 25589 euro**

Done Local intranet

Figure 2b

Figure 2a

```
constraint c1 not (Engine individual of Petrol18 and Styling="Elegance")
constraint c2 Engine individual of Diesel22 implies GearBox individual of Auto_transm
constraint c3 GearBox individual of Auto_transm implies Styling="Elegance"
```

Figure 2c

Figure 2. Configuration model of a Car (left) and the corresponding user interface (right)

Figure 2a illustrates the concepts. The component type `Car` has a part definition with part name `Engine`, allowed type `Motor`, and cardinality 1. The abstract component type `Motor` has concrete subtypes `Petrol18`, `Petrol20`, and `Diesel22`. `Car` also has a part definition with part name `GearBox`, allowed type `Transmission`, and cardinality 1. `Car` defines, out of the figure, a property with property name `Styling`, value type `string` constrained to values `"Standard"` and `"Elegance"`. In addition, `Car` specifies three constraints, shown in Figure 2c. `c1` specifies that `Engine` of type `Petrol18` and `Styling` `"Elegance"` are incompatible, `c2` states that `Engine` of type `Diesel22` requires a `GearBox` of type `Auto_transm`, and `c3` says that `GearBox` of type `Auto_transm` requires that `Styling` is `"Elegance"`.

PCML is declarative and it has formal implementation-independent semantics provided by mapping it to weight constraint rules[10], a form of logic programs. The basic idea is to treat the sentences of the modelling language as short hand notations for a set of rules in the weight constraint rule language (*WCRL*), see [3] for details. A configuration is a logical model (so-called *stable model*) of the set of rules representing the configuration model.

The class hierarchy and compositional structure are edited semi-visually in Modelling Tool. *Component type tree* displays the class hierarchy (top-left, Figure 2a) and serves as a starting point for adding, deleting and manipulating component types, their part and property definitions, and constraints. The compositional structure is shown in the *part hierarchy tree*, (bottom-left, Figure 2a). It enables adding allowed types and part definitions with drag&drop from the component type tree. The currently selected object is shown and edited to the right of the trees, the example in Figure 2a is part definition `Engine` in component type `Car`.

A web-based user interface for the end-user is generated without programming. The idea is that each selectable property or part of a component individual being configured generates a *question*. The modeller can define for a component type how the questions in an individual of that type are *grouped* and ordered. In the example of Figure 2a, parts `Engine` and `GearBox` of component type `Car` were put to the first group named "`Basic`", and property "`Styling`" was put to the second group "`Options`". Normally, questions of a group are answered before consequences are deduced, but the modeller can mark some questions as requiring immediate inference. Further, the modeller can give a display name in different languages to component types, parts, possible values in property domains, etc. Display names were given to subtypes of `Motor` and `Transmission` to offer information relevant for selecting a motor to English-speaking users, while using more technical names for component types in the configuration model. Grouping information, display names, etc., are stored as XML files.

3.2 WeCoTin Configuration Tool

WeCoTin enables users to configure products over the web using a standard browser. The component *ConfigurationServlet* (Figure 1) acts as a presentation layer that generates dynamically the user interface for end-users that employs HTML and JavaScript. The interface consists of the following parts: 1) The *configuration tree* (Figure 2b, left) gives an overview of the configuration: compositional structure is shown along with properties and their values. Effectively, selections already made and selections still to be made are shown. Error messages related to violated constraints are collected above the tree. The configuration tree also provides navigation to configure in free order: nodes of the tree serve as links to different parts of the configuration. 2) A status area (bottom-right in Figure 2b) indicates the status of the configuration in terms of consistency and completeness, e.g. a green OK means that the configuration is consistent and complete. The status area also shows calculation results, such as price. 3) A group of questions related to a component individual and derived from the configuration model and grouping information is represented as an HTML form (top-right in Figure 2b). Suitable HTML form elements are automatically selected for each question but the modeller can use Modelling Tool to specify the form elements to use. Incompatible alternatives are greyed out. However, the user is free to make incompatible selections. In this case, the user gets information about violated constraints. The form also has a number of buttons that are explained below. 4) Optional tailoring elements, e.g., a company logo or images can be added through separate HTML templates. The templates also define the layout of the user interface.

ConfigurationServlet provides a guided order for the configuration task by traversing the configuration tree a group at a time, in depth-first order, when the “Next”-button is used; clicking a branch in the tree provides a free order. The browser submits answers to the questions to ConfigurationServlet when the user presses the “Next”-button, navigates using the configuration tree, or answers a question marked as requiring immediate inference. ConfigurationServlet then modifies the configuration to reflect the answers and sends it to *ConfigurationServer* component (Figure 1). ConfigurationServer sends back a configuration reflecting the results of inference and calculations (described below). ConfigurationServlet generates a display with new questions and updated information reflecting the new configuration. When the configuration is consistent and complete, a “Ready”-button is shown enabling proceeding to order the product individual. This shows a summary of the configuration and “Order”-button that transfers the configuration to an exporter (described below).

The ConfigurationServer (Figure 1) provides the application logic layer of the configurator. It manages sessions with a number of simultaneous users, includes data structures to represent configuration models and configurations (*PCML core* in Figure 1) and provides a service to configure a product with respect to a configuration model and requirements. To provide configuration service, ConfigurationServer uses as the inference engine an implementation of the weight constraint rule language called *Smodels* [10]. The main functionality of the Smodels system is to compute for a WCRL program a desired number of stable models that are constrained by requirements specified as a so called *compute statement*. The Smodels system is based on a two-level architecture where in the first phase a front-end, *lparse*, compiles a WCRL program with variables into simple basic rules (BCRL) containing no variables. This potentially costly compilation process is performed off-line. The search for models of BCRL programs is handled using an efficient, linear space search procedure, *smodels*. Smodels is implemented in C++ and offers APIs through which it can be integrated into other software. Smodels is publicly available at http://www.tcs.hut.fi/Software/smodels/.

As a final step of modelling, the *Smodels interface* component in Modelling Tool (Figure 1) compiles [3] a PCML configuration model into a WCRL program, and further, using lparse (Figure 1), to BCRL. The BCRL form of the configuration model is loaded to smodels search procedure to repetitively configure a product. ConfigurationServer connects to smodels via its *Smodels interface* (Figure 1). Smodels interface translates the user requirements represented as property values and component individuals in a configuration to a compute statement that is sent through the API to smodels. Consistency of the requirements is checked by trying to compute a configuration that satisfies the requirements. Deducing consequences of requirements is based on computing an efficient approximation of the set of configurations satisfying the requirements. Intuitively, the approximation contains a set of facts that must hold for the configurations satisfying the requirements, a set of facts that cannot be true for the given requirements, and a set of unknown facts [10]. Based on this approximation, Smodels interface generates a new configuration, and hands it to ConfigurationServer. ConfigurationServer uses its calculation subsystem for computing results such as price or delivery time before the configuration is returned to ConfigurationServlet.

Our default exporter creates a XML representation of the configuration and stores the HTML summary of the configuration. An exporter interface provides an API for writing exporter modules to transfer configurations to external systems. Implemented exporters include Intershop 4 e-commerce system, EDMS2 product data management system and Vertex 3D CAD.

4 Empirical testing

In this section we describe a method for testing performance of configurators and apply it to WeCoTin. We briefly describe our benchmark products and summarise our empirical results. One could test a configurator by using real or randomly generated configuration models. There is a risk that random models without a large set of real products as a seed would not reflect the structured and modular nature of products designed by engineers. Therefore we test WeCoTin with real configuration models and a wide range of random requirements. For generating random sets of requirements, we consider how the configuration model appears to a 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 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 (*requirement items*) that a user could state. In our tests, a group is created for each property and part definition of the type of each individual. A value in the domain of a property and each potential part individual for a part corresponds to one requirement item. We generate random requirements by repetitively selecting at random a group and further a random requirement item of that group, while ensuring that at most as many items as is allowed by the configuration model are selected from each group. A *test case* contains a number of thus chosen requirement items. We note that this test methodology could be applied relatively easily to other formalisms because it only assumes the availability of a user requirements perspective.

We used PCML to model three screw compressor families and one 4-wheel vehicle. The compressor models were detailed almost to production quality, but the vehicle model originally created for demonstration purposes represented only about half of the sales view. In this paper we describe for brevity only one configuration model and its performance. Full test results of all products and further details of the test setup are described in [3] and are available at http://www.soberit.hut.fi/pdmg/Empirical/. A model called ESVS is reported because it shows the weakest performance in finding the first configuration satisfying the requirements and also because it is the largest model. There are 9 component types, 3 part definitions, and 24 properties, 17 of which have small domains of 2-3 possible values, but the largest domain size is 61. The number of constraints is 20.

We measure performance using execution time due to its practical importance for users. All the tests were run on a laptop computer with 1 GHz Mobile Pentium III processor. We used smodels with modifications that suppressed the output of found configurations to prevent the task from becoming I/O bound. According to our experiences, the timing result averages are repeatable to 1/10th of a second. The average time of 100 executions to translate the configuration model expressed in PCML to WCRL was 8.2 seconds. We generated 100 test cases for each even number of requirements up to the total number of groups. Table 1 shows the average performance of searching for ESVS configurations. A test case was considered *satisfiable* if a configuration was found with the requirements, otherwise it was considered *unsatisfiable*. Each row lists the number of requirements, the number of satisfiable cases, time to find one configuration and the time to determine unsatisfiability. "Find all" gives the average number of configurations per satisfiable case ("#cfgs /case") and the average rate of configurations found per second ("#cfgs / s"). All the results include the time required by smodels to read the BCRL program and to perform the required computation.

Table 1. ESVS compressor results with test cases

ESVS		Find first	Find all		Unsatisfiable
#requirements	#satisfiable	(seconds)	#cfgs / case	#cfgs / second	(seconds)
0	100	0,37	1,841,356,800	100066 (1 run)	-
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

5 Discussion and previous work

In this section, we first discuss the practical feasibility of WeCoTin in the light of what we consider the most important configurator-specific requirements: advanced support for long term management consisting of usability of the modelling language and support for product evolution, efficiency of the inference engine, and the basic architecture. We then discuss relationships of our approach to some of the most closely related work.

WeCoTin has not been in production use and its practical utility has thus not been fully proved. PCML contains an extension of product structures, which we find [7] a natural way for engineers to conceptualise products. PCML was expressive enough to model the intended sales views of the case products, and all its concepts were useful. However, adding separate modelling concepts for sales features or functions would improve modelling. Modelling the largest compressor and the vehicle required each a few hours from the first author, excluding knowledge acquisition. Furthermore, the language seems accessible to other engineers: a software developer of the WeCoTin project who had not participated in the development of PCML modelled two compressors, and a mechanical engineer with good understanding of configurable products but no computer science background modelled another type of a vehicle and further two compressors each with a few days of effort, including knowledge acquisition.

Informal feedback from manufacturing companies indicates that the web-based architecture enables easy access to the configurator for a significant part of their potential users connected to the Internet or Intranets, but would be a problem for some. It has been easy to integrate WeCoTin to external systems for demonstration purposes. The user interface and performance of the WeCoTin Configuration Tool has received positive feedback from several companies.

Our test results indicate that the inference engine performs well with the case products. There were no test cases with repeatable significantly inferior performance. The average configurations per second results weaken with an increasing number of requirements. However, this seems to be mostly illusory: because there are few configurations with many requirements, Smodels uses most of the time for reading the BCRL program and to set up the computation. Our case products had a relatively small number of components and properties. However, we feel that they are representative of what is needed in sales configuration. We expect that the suitability of the modelling method and good performance of our configurator also applies to other products suitable for web-based sales configuration.

PCML supports variation of the compositional structure in a way resembling generic product structures [11]. PCML adds object-oriented flavour, e.g. types, instances and inheritance but lacks separate concepts for mapping from input information to structure of the product individual. WeCoTin lacks support for modelling the evolution of products, components and their interdependencies. Therefore full support for advanced long-term management is insufficient, but such support is lacking also from reported previous configuration research.

There is a body of previous research in supporting configuration problems, e.g. [1] and [2], starting from the R1/XCON system at Digital. The main difference compared to XCON and AI shells such as OPS5, CleverPath Aion, or Blaze Advisor is that WeCoTin provides a declarative configuration modelling language aimed to configuration modelling. Further, this language has clear formal semantics, which provides provably sound and complete inferences based on the model and requirements. This is a general difference with respect to e.g. generic product structure based work [11] and most of the more recent work on configuration [2]. WeCoTin clearly separates modelling from the inference engine making it possible to improve the inference engine without changing the modelling language and its semantics. In XCON, both the product model and the control of the flow of inference were coded using unstructured production rules, which led to serious maintenance problems when the product changed, as the interactions between rules and their execution grew very complex [8]. Moreover, our configuration specific conceptual model makes automatic user interface generation practical.

However, WeCoTin currently supports a less expressive modelling language than some systems [2], e.g. resources and connections are missing making it difficult to model e.g. some rack-mounted products. Support for configuration tasks with dimensioning or other engineering calculations is limited as WeCoTin currently supports integer arithmetic only. Further, there are reports of configurators being used in real production environments, which indicates their practical utility. None of them have been built to specifically meet the requirements of the web-environment, however.

Parametric and feature based 3D CAD/CAE tools [12] are based on geometric modelling whereas WeCoTin configuration models do not include geometry. Inference in WeCoTin is sound and complete, which is not the case in typical CAD/CAE calculation and constraint systems. Therefore, a CAD/CAE user is eventually responsible for keeping the design consistent and must resolve situations where the tool does not find a solution even if one exists.

We provide detailed testing data on our approach unlike most of the previous work. Some of the notable exceptions are [4], [5], and [6]. Syrjänen configured Debian GNU/Linux operating system whose configuration model was expressed as packages, their versions and simple constraints. Syrjänen used an earlier version of Smodels [4]. Compared to this, we provide a richer modelling language geared towards mechanical products, while lacking support for version management. The configuration efficiency was approximately the same as in our largest ESVS model. Other previous work does not make direct performance comparison possible due to missing details and differences in modelling. Sharma and Colomb developed a constraint logic programming based, port and connection oriented language for configuration and diagnosis tasks. Experimental results stem from thin ethernet cabling configuration. Finding a 12-node configuration including 126 port connections required 12 seconds on two 60 Mhz SuperSparc processors [5]. Mailharro used the Ilog system to configure the instrumentation, control hardware and software of nuclear power plants. The modelling language is richer and the case product larger and more complex than ours. Several thousand component individuals were created and interconnected in about an hour of execution time on a Sun Sparc 20 [6].

6 Conclusions and future work

We presented the requirements and design of a novel configurator prototype that supports tailoring a configurable product. The tool contains a semi-visual modelling tool based on a high-level object- and product structure-oriented modelling language with a clear formal semantics. In addition, it contains a configuration support tool whose architecture is aimed at e-commerce. This tool provides intelligent support for configuring a product by applying a state-of-the-art inference engine for the form of logic programs on which the formal semantics is based. The tool creates automatically a web-based user interface. We modelled four real products from two different domains and tested the efficiency of the inference engine on them. For this, we used a modelling-language-independent method for systematic testing based on simulating a naïve user inputting random requirements to a configurator.

On the basis our experiences, the prototype is suitable for e-commerce. The modelling language allows efficient modelling of products for web-based sales configuration and seems suitable for engineers without programming or artificial intelligence background. The inference engine appears to be efficient enough for practical use. However, the relatively small sample of products from only two domains was modelled by the developers of the system. Thus, more research is required to validate if the modelling language is usable by product engineers and managers, and if it is suitable for products from different domains. Further, the language should be extended to cover more complex domains. One topic identified in this work is to provide support for modelling sales features. Moreover, the efficiency of the inference engine should be tested on products that are larger and potentially more computationally costly to configure, e.g. telecommunications equipment modelled from an engineering point of view. Finally, support for the management of product evolution should be added.

ACKNOWLEDGEMENTS

This work has been supported by Technology Development Centre of Finland and Academy of Finland (third author, project 53695). We thank Andreas Anderson, Mikko Heiskala, Asko Martio, Juha Nurmilaakso, Mikko Pasanen, Hannu Peltonen, and Kati Sarinko for their valuable efforts. Finally, we thank Gardner Denver Oy for sharing the configuration knowledge.

REFERENCES

[1] Sabin D. and Weigel R., "Product configuration Frameworks—a survey", IEEE Intelligent Systems & Their Applications, Vol 13(4), 1998, pp.42-49.

[2] Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM), Special issue in configuration, Vol. 12, 1998.

[3] Tiihonen J., Soininen T., Niemelä I. and Sulonen R., "Empirical Testing of a Weight Constraint Rule Based Configurator", Papers from the 4th Configuration Workshop, 15th European Conference on Artificial Intelligence (ECAI-2002), 2002, pp.17-22.

[4] Syrjänen T., "Including Diagnostic Information in Configuration Models", Proceedings of the First International Conference on Computational Logic, 2000.

[5] Sharma N. and Colomb R., "Mechanising Shared Configuration and Diagnosis Theories Through Constraint Logic Programming", Journal of Logic Programming, Vol. 37, 1998

[6] Mailharro D., "A classification and constraint-based framework for Configuration", AI EDAM, Vol. 12, 1998, pp.383-397.

[7] Tiihonen J., Soininen T., Männistö T. and Sulonen R. "State-of-the-practice in product configuration—a survey of 10 cases in the Finnish industry. " In Tomiyama T., Mäntylä M. and Finger S., editors, Knowledge Intensive CAD, Vol 1, Chapman & Hall, 1996.

[8] McDermott J. "R1 ("XCON") at age 12: Lessons from an Elementary School Achiever", Artificial Intelligence, Vol 59, 1993, pp.241-249.

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

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

[11] Erens, F. The Synthesis of Variety - Developing Product Families, PhD Thesis, University of Eindhoven, the Netherlands, 1996

[12] Shah J, and Mäntylä M. Parametric and Feature-based CAD/CAM: Concepts, Techniques and Applications, John Wiley and Sons, 1995.

For more information please contact:
Juha Tiihonen, SoberIT, Helsinki University of Technology, P.O. Box 9600, 02015 HUT, Finland
Tel: +358 9 451 3242 Fax: +358 9 451 4958 E-mail: Juha.Tiihonen@hut.fi URL: http://www.soberit.hut.fi/~jti/