AI Communications 26 (2013) 99–131
DOI 10.3233/AIC-2012-0547
IOS Press

WeCoTin – A practical logic-based sales configurator

Juha Tiihonen [a,*], Mikko Heiskala [a], Andreas Anderson [b] and Timo Soininen [a]

[a] *School of Science, Department of Computer Science and Engineering, Aalto University, Aalto, Finland*
E-mails: {juha.tiihonen, mikko.heiskala, timo.soininen}@aalto.fi
[b] *Variantum Oy, Espoo, Finland*
E-mail: Andreas.Anderson@variantum.com

Abstract. Configurable products can realize the ideal of mass-customization by satisfying individual customer requirements efficiently. IT support provided by configurators enables adapting such products for individual customers efficiently and without errors. Few of numerous configurators have been evaluated with respect to modeling efficacy and performance on several product domains, and few evaluation methods exist. Applying the Design Science method, we describe and evaluate a novel configurator called WeCoTin. WeCoTin is based on a high-level object oriented modeling conceptualization and corresponding modeling language with clear formal semantics. WeCoTin consists of a semi-visual Modeling Tool and a web-based Configuration Tool. It applies an inference engine that follows the logic-based answer set programming paradigm. A way to characterize configuration models is proposed and applied to characterize over 20 real-world configuration models, and to evaluate utility of modeling mechanisms. Furthermore, performance is evaluated with real-world products using a developed method, and found adequate.

Keywords: Knowledge-based configuration, evaluation of configurators, configuration knowledge characterization, integrated development environment

1. Introduction

1.1. Background

Mass-customization [34] aims to provide products or services that closely match the individual needs of customers while retaining mass-production-like efficiency. Mass-customizers need three fundamental capabilities [37]. First, solution space development is required to identify the product attributes for which customer needs diverge. Second, a robust process enables reuse or recombining of existing organizational and value-chain resources to fulfill a stream of differentiated customer needs. Finally, choice navigation capability is required to support customers in identifying their own solutions while minimizing complexity and the burden of choice.

Business based on configurable products is one way of achieving mass-customization. A *configurable product* is the result of solution space development, an in advance provided design of a family of products that can be adapted to meet customer requirements within the scope of designed variability. A *configuration task* produces a specification of a *product individual* that meets customer requirements and conforms to rules of the configurable product. A specification of a product individual based on a configurable product is called a *configuration*. The product family description containing all the information on the possibilities of adapting the product to customer needs is called a *configuration model.* A configuration model specifies the entities that can appear in a configuration, their properties, and the rules on how the entities and their properties can be combined. The configuration model is an abstraction of a real-world product family specifically constructed for configuration purposes [57].

There are significant potential benefits and challenges related to mass-customization and configurable products, as summarized in [18]. Practical challenges include incomplete or inconsistent configurations created during the sales process, which causes lengthy iterations in the sales-delivery process and disproportionate quality control costs. Lead times in the config-

*Corresponding author: Juha Tiihonen, School of Science, Department of Computer Science and Engineering, Aalto University, P.O. Box 19210, FIN-00076, Aalto, Finland. E-mail: juha.tiihonen@aalto.fi.

0921-7126/13/$27.50 © 2013 – IOS Press and the authors. All rights reserved

uration process may be excessive, and many practical configuration models are poorly documented, incomplete, difficult to understand and outdated. Many of the challenges can be addressed with Information Technology (IT) support due to the well-defined nature of the configuration task.

1.2. Configurators

A class of systems, *configurators*, makes it possible to represent the variability of configurable products by creation and management of configuration models and to support users in performing the configuration task. Customer needs are represented as requirements, and variability of the product is captured in a configuration model. Based on requirements and the configuration model, the configuration engine produces a configuration, which specifies a product individual that meets the customer's needs and that can be delivered to the customer. Thus, a configurator can form a basis for the choice navigation capability of a company.

Numerous configurators have been developed both as research prototypes and as commercial software. The landmark R1/XCON was deployed at Digital Equipment Corporation in early 1980s [23], and experiences, benefits and challenges of using it are widely documented, see, e.g., [24].

Major research efforts have been undertaken to develop configurators applicable to solving general configuration tasks instead of a specific domain, e.g., [4,13,47]. A large number of commercial general-purpose knowledge-based configurators exist; Anderson [1] identified 30 vendors based on their web pages. In addition, it is a norm that prominent enterprise resource planning systems include a configurator module, e.g., [6,14].

Configurators are deployed relatively widely. For example, 580 web-based configurators were listed in the International Configurator Database [5].

1.3. Problem solving in configurators

Numerous problem solving methods have been applied to configuration tasks, and several overviews exist. In their taxonomy of the types of problem solving methods used for design and configuration, Wielinga and Schreiber [62] divide *configuration problem solving methods* to *knowledge-intensive methods* and *uniform methods*. Uniform methods apply the same reasoning methods to all problems, while knowledge-intensive methods use explicitly modeled knowledge to constrain and direct the problem solving. Knowledge-intensive methods (*propose, critique, and modify; case based;* and *hierarchical*) are not considered further in this work, because uniform methods are mature enough to support the configuration tasks in the sales configuration of many products or services.

Uniform methods include *constraint solving* and *logic-based methods*. Constraint satisfaction (CSP) and its extensions have gained significant popularity [12,22,26]. Many consider constraint-based methods ideal for solving configuration problems, e.g., Gartner Group and Haag [7,16]. Several logic-based methods have been applied to configuration, including direct programming in Prolog or through a higher-level modeling layer, see, e.g., [38]. Description logics (DLs), e.g., [2,25], and constraint logic programming have also been applied [39]. Weight Constraint Rules have been applied to directly model configuration problems in research systems [49], and a method has been proposed to translate configuration domain modeling concepts into Weight Constraint Rules [43].

1.4. Configuration knowledge modeling

Many problem solving methods provide an explicit or implicit conceptual model for representing configuration models, requirements, and configurations, i.e., configuration knowledge. A straightforward approach to configuration knowledge modeling is to express knowledge directly in the language of the problem solving method to which the configurator applies. Rule-based approaches, constraint satisfaction and its dynamic extensions, several logic-based approaches, and different formalisms of propose-and-revise methods have been applied. For summaries, see [36,45].

We argue that a more practical approach in terms of configurator maintainability is to provide a configuration-specific modeling conceptualization that can be applied to express configuration knowledge. The task of the configuration system is to map this knowledge into a form that can provide the inference required to support the configuration task.

Several configuration domain-specific conceptualizations exist that are independent of problem solving methods. These can be roughly classified as *connection-based* [27], *resource-based* [17], *structure-based*, e.g., [4] and *function-based* [28] approaches. The conceptualizations have little in common, other than the central notion of a component. A synthesis and generalization of the previously mentioned approaches has been presented [44]. A similar synthesis based on more

concrete representation that employs the Unified Modeling Language (UML) [35] with specific stereotypes and Object Constraint Language (OCL) has been proposed, e.g., [10].

1.5. Research questions, method and outline of the paper

This paper describes and evaluates a configurator implementation, claiming contributions that justify yet another scientifically relevant configurator construction effort.

We believe that the synthesis of [44] covers a broad range of configuration problems. However, in many companies that we visited during several research projects, order forms for sales configuration were relatively simple, typically a page or two in A4 format. This relative simplicity of the methods used in practice raised several questions. Would it be the case that many elements of the generalized conceptual model are needed only in complex configuration tasks? Could a significantly reduced subset of the conceptualization support a practical and relevant set of sales configuration tasks? Would it be useful and practical to include only compositional structure, taxonomic concepts (classification and inheritance), attributes, and constraints? Do these concepts provide benefits for modeling?

In our view, the idea of translation of configuration knowledge into Weight Constraint Rules (a form of logic programs) [43] provides an interesting basis that deserves empirical validation of feasibility as an alternative to the nearly dominant constraint-solving based approaches. Given an efficient implementation of Weight Constraint Rules [30], would it be feasible to build a configurator that provides a high-level conceptual model for the modeler and applies this alternative problem solving method?

Most configurators have been described and evaluated in the context of individual domains, the characterizations of configuration models are relatively thin, and runtime performance has been reported in ad-hoc manner, if at all. This makes it difficult to evaluate the applicability of general purpose configurators to various domains. Could some light be shed on this aspect?

This work aims answer the following research questions:

- Q1: Is a function-oriented subset of a configuration conceptualization [44] useful for modeling configuration knowledge?
- Q2: Can the translation of configuration knowledge into Weight Constraint Rules [43] provide a practical and feasible basis for a product configurator?
- Q3: Can industrially relevant configuration problems be effectively modeled and configured with a configurator constructed based on the conceptual model of Q1 and the logic-based approach of Q2?
- Q4a: How should configuration models be characterized? Q4b: What modeling mechanisms are useful? Q4c: How should performance of a configurator be evaluated?

We approach these research questions through the *Design Science* approach [19], a research method that creates and evaluates IT artifacts intended to solve identified organizational problems. Design Science research is conducted through *building* and *evaluation* of *artifacts* designed to meet the identified business need, with the ultimate goal being utility. Artifacts can be *constructs* (vocabulary and symbols), *models* (abstractions and representations), *methods* (algorithms and practices), or *instantiations* (implemented or prototype systems). Evaluation of an artifact often leads to the need to refine it. Contributions are assessed through applying them to the identified business need in the appropriate environment.

In Fig. 1 the path of research is explained. Rectangles and rounded rectangles representing units of research in Fig. 1 are referred to as *shapes*; each has a numeric identifier. It was necessary to acquire an understanding of the business context of configurators (shape 1): interviews in 10 companies provide working knowledge on configuration-related process practices and problems, a general understanding of configurable products and their long-term management, and a description of models and tools in use [52,56,58]. This foundation enabled an understanding of the requirements for configurators (shape 2). An additional contribution to understanding requirements comes from experiences in building a single-purpose configurator prototype with an "Intelligent CAD" tool [51] and close co-operation with half a dozen other companies that used or planned to use configurators. To save space, only a summary of main requirements is discussed; see Section 2.

A generalized conceptualization for configuration knowledge [44] (shape 3) provides a conceptual basis for this work. The Product Configuration Modeling Language (PCML) (shape 4, see Section 3.3) [33] gives a syntax for a subset of the conceptual model. It

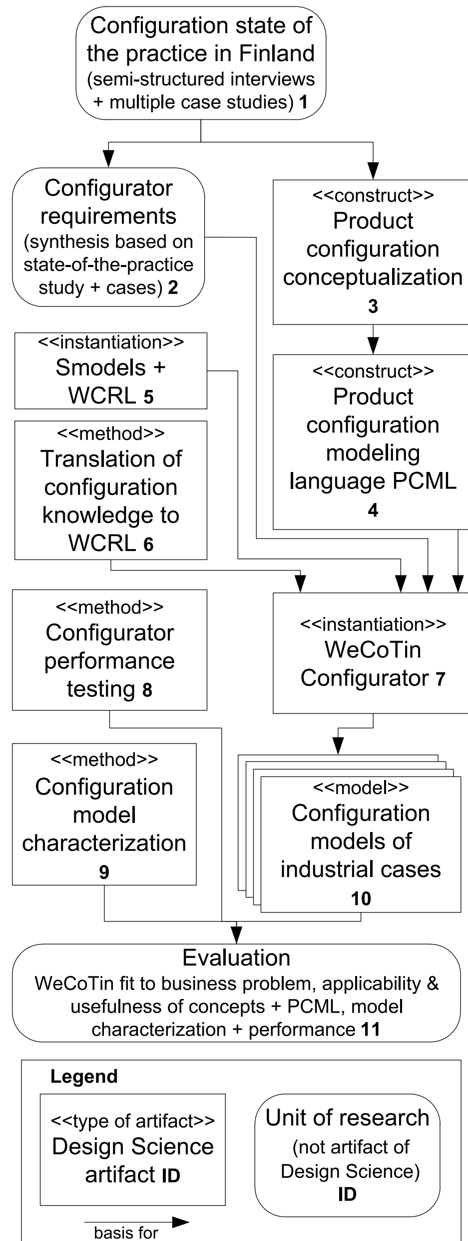

Fig. 1. Units of research, artifacts and their dependencies.

includes taxonomy, compositional structure, attributes and constraints. Semantics of PCML are provided by mapping PCML to Weight Constraint Rule Language (WCRL) (shape 6), applying the idea of [41]. The Smodels system (shape 5) provides an efficient inference engine for WCRL [30,50]. Development of WCRL and Smodels took into account configuration as a potential application domain.

The main construct in this work, a domain-independent configurator called the WeCoTin configurator (shape 7), is described in Section 3, along with a running example. An overview of WeCoTin is given in [60]. Direct foundations of WeCoTin include work described above (shapes 3–6).

WeCoTin is evaluated in Section 4. Applicability of WeCoTin and its modeling capabilities to industrial problems are verified by modeling and configuring the sales view of 22 real-world products or services, 14 in a complete way and 8 partially (shape 10). These and some additional configuration models are characterized by size and application of modeling constructs [54], summarized in Section 4.1. Experiences with modeling capabilities are described in Section 4.2. Run-time performance of WeCoTin is evaluated with a developed method (shape 8) for performance testing of configurators based on real-world configuration models and random requirements [59], as discussed in Section 4.3. As a whole, evaluation of WeCoTin (shapes 10, 11) is more multifaceted than any other general purpose configurator of which the authors are aware.

Discussion in Section 5 analyzes previous work, provides results of this work, and identifies future work. Finally, Section 6 points out answers to research questions and represents conclusions.

2. Configurator requirements

In this section, we present central requirements specific to a practical web-based configurator. The requirements were identified in joint projects with the manufacturing industry and our previous work, e.g., [56].

A *modeler* creates and maintains configuration models and related information, and an *end-user* (or *user*) configures a product.

Products evolve over time as new features are introduced and designs are improved or corrected. Product evolution inevitably leads to new configuration model versions. Long-term management of configuration models has often been problematic; an extreme example is the R1/XCON system [24].

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 who are traditionally needed to maintain configurators and eliminates the error-prone communicating of product knowledge to separate modelers who are not product experts. Mod-

eling should be easy for product experts to understand and *declarative*, allowing the modeler to specify *what* kind of product individuals are valid, instead of *procedural*, requiring specification of *how* to create them. The modeling language should be object-oriented to divide configuration models into relatively independent pieces with low complexity and to exploit their common characteristics. Furthermore, the modeling language 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 modeling, advanced long-term management requires support for modeling the evolution of products, components, and their interdependencies in a way that resembles configuration management (CM) and product data management (PDM). Moreover, it should be possible to efficiently deploy configuration models to salespeople and customers without the risk of using outdated versions, 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 fully deducinges 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 consistent configuration. In addition, explanations for incompatibility of selections should be available. This helps users learn 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 the 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.

Finally, an interactive configurator should provide adequate performance in terms of length and predictability of response time. According to [29], about 0.1 s is the limit that allows the user to feel that the system is reacting instantaneously, and about 1 s is the limit for the user's flow of thought to stay uninterrupted.

3. WeCoTin configurator

In this section, we describe the web-based configurator prototype *WeCoTin* (an acronym for Web Configuration Technology). We show how its high-level architecture and main functionality meet the high-level requirements on practical configurators identified above, and we describe the subset of the conceptual model of [44] that has been implemented. We also show how the conceptualization was extended with practical and necessary constructs such as default values, pre-selection packages, price and delivery time determination mechanisms.

WeCoTin consists of two main components: a graphical modeling environment *The Modeling Tool* (Fig. 2, right) and the web-based *Configuration Tool* that supports the configuration task (Fig. 2, left). These components are detailed after presentation of a running example.

WeCoTin is implemented using the Java 2 Platform and Java programming language, except for the component Inference Engine, which consists of smodels and lparse programs of the Smodels system that are implemented in C++, and user interface components that employ some JavaScript a.k.a. ECMAScript [8].

3.1. Running example

A subset of variation offered by a real car form the basis for a running example.

- The basic variation concerns the Motor and Transmission.
  - The *Motor* of the car is selected from 5 types – a 2.0, 2.5, or 3.0 l petrol engine, or a 2.5 or 3.0 l diesel engine.
  - The type of *Transmission* is 'Manual 6-speed', 'Sequential manual', or 'Automatic'.

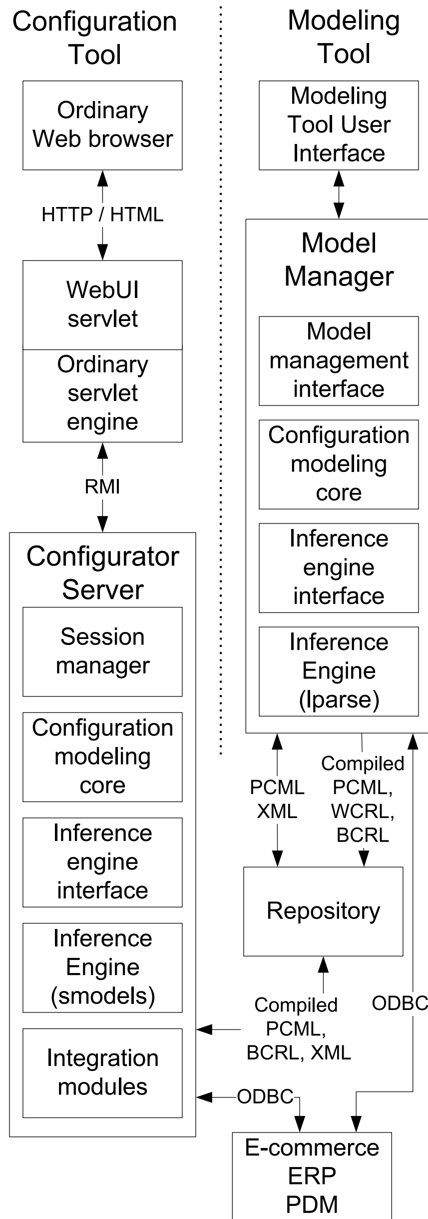

Fig. 2. WeCoTin architecture overview: Configuration Tool on the left and Modeling Tool on the right.

- Safety and Security equipment has the following two variation points:
  - *Headlights* are either 'standard' or 'Bi-Xenon' type. The Bi-Xenon Headlights can optionally be specified as 'Adaptive Headlights'.
  - *Headlight Washer System* is an optional feature that can be included or left out.
- Comfort equipment has two variation points:
  - *A Cruise Control System* is always included, but the variation to be selected is either 'standard' or 'active'.
  - A *Navigation System* is optional – the user can specify a car without a Navigation System, or select a 'Business Navigation System' or a 'Professional Navigation System'. A Navigation System comes with an electronic map that covers Finland, Germany, Spain, Sweden, or the United Kingdom. The *Navigator Map* area should only be selected if ordering a navigator. In the context of 'Professional Navigation System', the user must specify if an optional *voice input* is wanted.
- Accessories are included for demonstrating higher cardinalities:
  - *Accessories* of types 'A1', 'A2' and 'A3' are available. One, three, or four accessories are to be selected for each car.

A number of constraints restrict the set of consistent configurations:

- 'Manual 6-speed' Transmission is incompatible with 3.0 l Motors.
- 'Active' Cruise Control requires 'Bi-Xenon' Headlights.
- 'Adaptive Headlights' requires 'Headlight Washer System'.

In addition, prices are specified for most selections. Listing 1 represents the corresponding configuration model in PCML (PCML is discussed in Section 3.3).

Listing 1
Sample configuration model in PCML

```
01 configuration model WecotinCar
02
03 feature WeCoTinCar
04 attribute Motor value type string
05 constrained by $ in list ("20i", "25i",
06 "25d","30i","30d") no default
08
09 attribute Transmission value type string
10 constrained by $ in list("Automatic",
11 "Sequential_Manual","6-speed")
12
13 attribute Cruise_control value type
14 Boolean proper default false
16
17 subfeature Headlights allowed features
18 BiXenon_Headlights, Standard_headlights
19 cardinality 1
20 proper default @Standard_headlights
22
```

```
23 attribute Headlight_Washer_System
24 value type Boolean proper default false
26
27 subfeature Navigator allowed features
28 Navigation_System cardinality 0 to 1
29 proper default none
31
32 subfeature Accessories allowed features
33 Accessory cardinality 1,3 to 4
34 proper default @A1(Attr_a1=true)
36
37 constraint
38 Manual_6_speed_Incompatible_with_3Litre
39 not ((Transmission = "6-speed" ) and
40 ((Motor = "30i") or (Motor = "30d")))
41
42 constraint
43 Active_Cruise_Control_Requires_BiXenon_
Headlghts
44 ( Cruise_control = true ) implies
45 ($config.Headlights individual of
46 BiXenon_Headlights)
47
48 feature BiXenon_Headlights
49 attribute Adaptive_headlights
50 value type Boolean proper default false
51
52 constraint Adaptive_Headlights_Require_
Headlamp_Washer_System
53 ( Adaptive_headlights = true ) implies
54 ($config.Headlight_Washer_System=true)
55
56 feature Standard_headlights
57
58 feature Navigation_System abstract
59
60 attribute Navigator_Map value type string
61 constrained by $ in list("Finland",
62 "Germany","Spain","Sweden","United
Kingdom")
63
64 feature Business_Navigation_System subtype
65 of Navigation_System
66
67 feature Professional_Navigation_System
68 subtype of Navigation_System
69
70 attribute Voice_navigation_input
71 value type Boolean proper default false
72
73 feature Accessory abstract
74
75 feature A1 subtype of Accessory
77 attribute Attr_a1 value type Boolean
79
80 feature A2 subtype of Accessory
82
83 feature A3 subtype of Accessory
85
86 configuration feature WeCoTinCar
```

3.2. WeCoTin Configuration Tool

WeCoTin enables users to configure products over the web using a standard browser. A web-based system architecture was selected for several reasons. Suitability for e-commerce of configurable products is an important design criterion. Thus, end-users should be able to access the configurator through the Internet, which makes stand-alone, separately installed applications impractical. Multiple simultaneous users and multiple simultaneous configuration models are supported on the server side.

An important problem of companies without a configurator is ensuring that up-to-date configuration models and prices are used when selling products [58]. Because WeCoTin is accessed through the web, it provides centralized configuration model management without the need to replicate or otherwise arrange configuration models to be available in local computers. WeCoTin's web-based architecture makes it possible to deploy changes by updating a central database without the need to replicate information on client computers. When a server's product and price database is updated, it is immediately available for use.

The component *WebUI servlet* (Fig. 2, upper-left) acts as a presentation layer that dynamically generates the user interface for end-users, employing HTML and JavaScript. The interface consists of several parts; these are indicated with a letter and description in Fig. 3. The *configuration tree* (Fig. 3, B) gives an overview of the configuration: compositional structure is shown, along with attributes and their values. Selections already made and selections still to be made are shown, and links facilitate a free order of making selections. The status area (top-left, Fig. 3, C) indicates the status of the configuration in terms of consistency and completeness, and shows calculation results such as price and estimated delivery. The three possible states of a configuration and corresponding symbols are shown in bottom right, Fig. 3, D.

A group of questions related to a product individual, derived from the configuration model and user interface generation information (see Section 3.4) is represented in the Question area (Fig. 3, A). The `6-Speed Transmission` in Fig. 3 is incompatible with current selections. The user is free to make incompatible selections. This informs users of what is available in the configuration model, and does not prevent users from selecting an interesting alternative, even if previous choices are incompatible with the desired alternative.

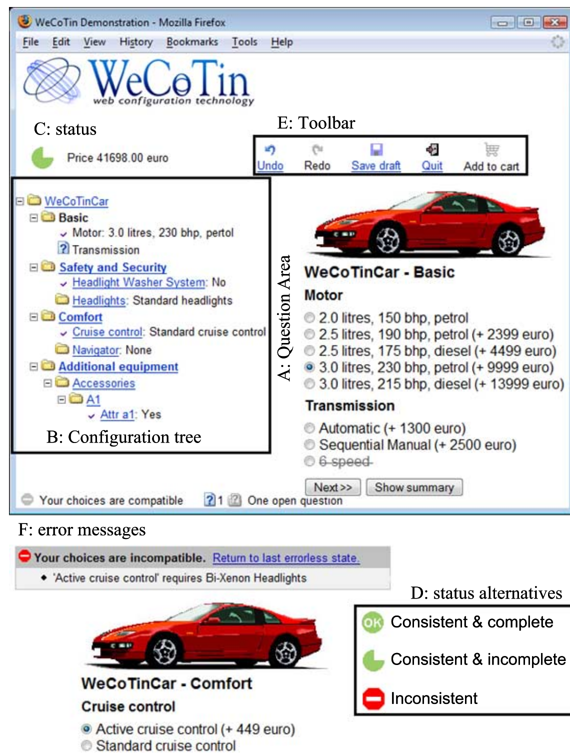

Fig. 3. WeCoTin Configuration Tool user interface for end users. A: questions to answer and wizard-style 'Next' button; B: Configuration tree gives an overview and free order of navigation; C: status, usually price and one of alternatives in D; E: toolbar for other actions. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-2012-0547.)

The user is informed about inconsistencies by collecting error messages of violated constraints on top of the Question area, area F of Fig. 3 shows an example. The continuous display of error messages relieves burden on user's memory compared to pop-ups.

Configuration in a wizard-style pre-determined order is available through the "`Next`" – button at the bottom of the Question area. It traverses the configuration tree in pre-order. The "`Show summary`" – button displays a summary page. The toolbar (Fig. 3, E) provides actions such as multi-step undo and redo, proceeding to buying a consistent and complete configuration (shopping basket icon), saving a configuration, and for terminating the configuration session.

3.3. The Modeling Tool and configuration modeling

The Modeling Tool is used for creating and editing configuration models and information needed to generate a user interface for end-users. The "Modeling Environment User Interface" in the Modeling Tool (Fig. 2) displays the current configuration model and facilitates editing it. Data structures representing the configuration model in the "Configuration modeling core" are modified through the "Model management interface" as the result of modeler's actions.

Configuration models are expressed in *Product Configuration Modeling Language (PCML)* described below. A save operation stores the configuration model as PCML. The tool also compiles the configuration models for use in WeCoTin Configuration Tool by utilizing the "Inference Engine interface" and the compilation component lparse in the "Inference Engine". This creates additional compiled representations of the configuration model, discussed in Section 3.5.

The following subsections describe the modeling language used to represent configuration models, and aspects of the Modeling Tool that facilitate semi-graphical editing of configuration models without the need for knowing the syntax of the modeling language. We follow a pattern where the first subsection describes central modeling language concepts, and the following subsection describes the corresponding Modeling Tool support.

3.3.1. PCML overview

Modeling in WeCoTin is based on a function-oriented subset of the configuration knowledge ontology [44]. *Functions* of the ontology are called *features* in the implementation. For historical reasons, the original WeCoTin implementation used the concept *property* to designate attributes. Originally, WeCoTin was developed with component-oriented modeling instead of feature-oriented. The decision to focus WeCoTin purely on sales configuration caused the renaming of concepts. Feature types were called *component types* and *subfeatures* were called *parts*.

PCML, Product Configuration Modeling Language [33], is used in the WeCoTin configurator as the language for representing configuration models. PCML is object-oriented, declarative, and has formal implementation-independent semantics. The semantics of PCML are provided by mapping it to Weight Constraint Rules [43]. The basic idea is to treat the sentences of the modeling language as shorthand notations for a set of sentences in the Weight Constraint Rule Language (*WCRL*). However, some important extensions to the conceptualization, most notably defaults, do not have formal semantics.

The main concepts of PCML are *feature types*, their *compositional structure*, *attributes* and *constraints*. Feature types define the subfeatures (parts) and at-

tributes of their *individuals* that can appear in a configuration.

A configuration is always based on a particular configuration model, and it represents the current (possibly incomplete or inconsistent) state of the configuration session. A configuration is a non-empty tree of feature individuals. Each feature individual is a direct individual of a single feature type. There is exactly one feature type (*configuration type*) whose only individual (the *configuration individual*) serves as the root of the compositional structure.

3.3.2. Taxonomic hierarchy and attributes in PCML

Feature types are organized in a *class hierarchy* where a *subtype* inherits the attribute and subfeature definitions of its *supertypes*. A predefined *root feature type* with the name 'Feature' serves as the root of the class hierarchy. A feature type is either *abstract* or *concrete*. Only an individual directly of a concrete type can be used in a configuration. Multiple inheritance among feature types is allowed.

A feature type may define attributes that parameterize or otherwise characterize its type. An *attribute definition* consists of an attribute *name*, an attribute *value type*, and *a necessity definition* indicating if the attribute must be assigned a value in a complete configuration. Base types of supported value types are Boolean, integer, and string with provisions for floats. Through *necessity definition* (or refinement, discussed below), an attribute is *obligatory*, *optional*, or *forbidden*. In a complete configuration, optional and obligatory attributes have a value, or a special value `none` that designates an explicit assignment of no value to an optional attribute. A value cannot be assigned to a forbidden attribute.

When a type inherits data from a supertype, the type can use the inherited data as such or it can modify the inherited data by means of *refinement*. Refinement is semantically based on the notion that the set of potential valid individuals directly of the subtype is smaller than the set of valid individuals directly of the supertype. *Optionality Refinement* can explicitly specify an optional attribute as obligatory or forbidden. *Value Type Refinement* restricts possible values, *Fixed Value Refinement* gives a fixed value to an attribute, and a *Default Value Refinement* specifies a different default value.

There are three ways for a feature individual to have a value for an attribute. First, if a feature type defines a *fixed value* for an attribute, all direct and indirect individuals of the feature type have this value for the attribute. Second, the individual can *assign* a value to the attribute. Third, a feature type can specify a *default value*. If there is no fixed value or assignment, the default value is used, unless maintaining consistency of the configuration requires (possibly temporary) removal of the default value.

Constants and *value types* can be defined. For example, attributes that share a domain could all use a common value type. Value types whose base value type is integer can be flexibly constructed with individual values, ranges, enumerated lists, constants, and tests based on functions. String value types are based on enumeration; free domains are possible, but such attributes are not usable in inference.

In the running example, lines 58–62 in Listing 1 define an abstract feature type `Navigation_System` with attribute `Navigator_Map`. It has concrete subtypes `Business_Navigation_System` (lines 64–65), and `Professional_Navigation_System` (lines 67–71) that inherit the attribute from their supertype. Further, `Business_Navigation_System` defines a new Boolean attribute `Voice_navigation_input`.

3.3.3. Taxonomic hierarchy and attributes in the Modeling Tool

The Modeling Tool provides full editing of feature types, their class hierarchy, and attributes. This includes creating, removing, renaming (maintaining model consistency) and changing of subclass hierarchies. The *Feature type tree* displays the class hierarchy (Fig. 4, A) and serves as a starting point for editing all aspects of the types. A type that is a direct subtype of several types (multiple inheritance) would appear in the tree as a subtype of all its direct supertypes. Editing the selected object takes place in the right side of the display or sometimes in pop-up windows. In the Feature type tree (Fig. 4, A), colored squares differentiate between concrete (blue) and abstract (red) types and the configuration type (green).

Feature type overview (Fig. 4, C) allows adding or removing attributes, and changing the concreteness or the name of the currently selected type. The tab `Attributes` shows an overview of the attributes of the selected feature type (`WeCoTinCar`). Special support is provided for defining enumerated attributes. The enumeration attribute editor in Fig. 4, D displays lines 4–6 of the running example.

3.3.4. Compositional structure in PCML

A feature type defines its compositional structure through a set of *subfeature definitions*. A subfeature definition specifies a *subfeature name*, a non-empty set

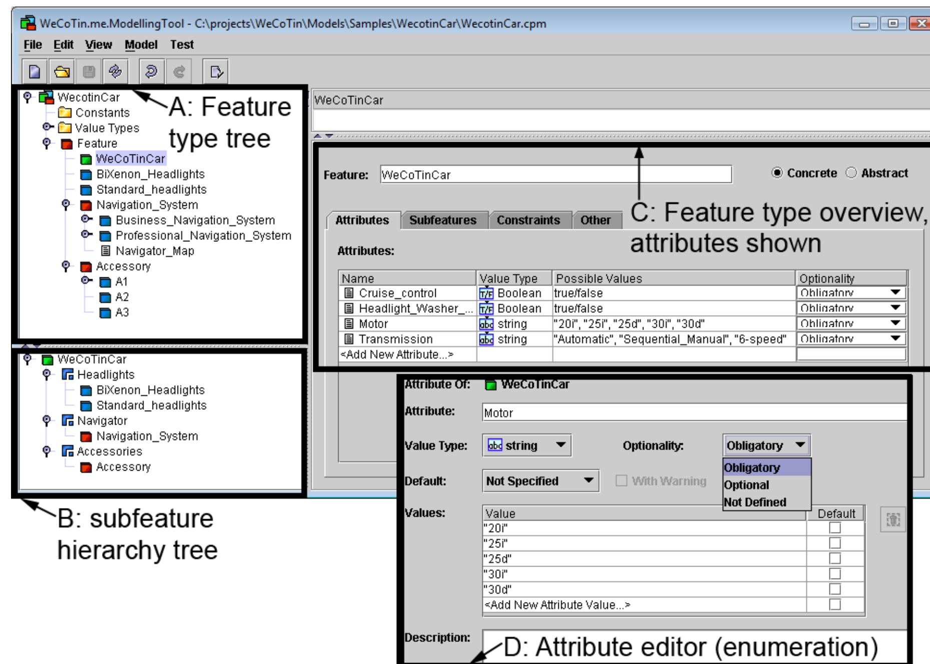

Fig. 4. Configuration model in the Modeling Tool showing attributes of feature type `WeCoTinCar` and the enumeration attribute editor for attribute `Motor`. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-2012-0547.)

of *possible subfeature types* (*allowed types* for brevity) and a *cardinality* indicating the valid number of subfeatures.

In a configuration, subfeatures (parts) of a feature individual are *realized* with feature individuals. The realizing feature individual(s) "fill the role" created by the effective subfeature definition. We say that feature type X has subfeature definition with part name P, with allowed types T, cardinality C, and (optionally) a default/fixed realization D. The cardinality is specified as a set of non-negative integers or integer ranges (e.g., `1, 3 to 5`). The realization of a subfeature cannot be valid unless the number of feature individuals realizing a subfeature is a member of the effective cardinality of the subfeature. For example, subfeature with name `Navigator` of feature individual `WeCoTinCar_1` could be realized by a feature individual `Business_Navigation_System_1` of type `Business_Navigation_System`. In other words, `WeCoTinCar_1` has `Business_Navigation_System_1` as part with subfeature name `Navigator`. An explicit empty realization is possible, when cardinality 0 is allowed. An "optional part" common in industrial products is modeled as a subfeature that allows an empty realization. A default realization can include attribute values and subfeature realizations of arbitrary depth. In addition, an empty realization (special value `none`) can be set as the default realization. Inherited subfeatures can be refined by restricting allowed types, cardinality, or by modifying or hiding default subfeature realizations.

Similar to attributes, there are three ways for a feature individual to have a realization for a subfeature. First, a *fixed realization* applies to all direct and indirect individuals of the feature type. Second, a realization can be *assigned* to the subfeature. Third, a *default realization* can apply. During the configuration process, it is possible that a subfeature does not have any realization. However, in a complete configuration, all subfeatures must be realized.

3.3.5. *Compositional structure in the Modeling Tool*

The compositional structure is shown in the *subfeature hierarchy tree* (Fig. 4, B). Drag & Drop from the feature type tree enables adding allowed feature types and subfeature definitions. An overview of the subfeatures of the active type `WeCoTinCar` is shown in Fig. 5, A; the tab 'Subfeatures' is a part of the Feature type overview shown in Fig. 4, C.

In the running example, lines 32–34 specify that feature type `WeCoTinCar` has a subfeature definition with subfeature name `Accessories`, allowed feature type `Accessory`, and cardinality `1, 3 to 4`. The definition is shown open in (Fig. 5, B). When active in

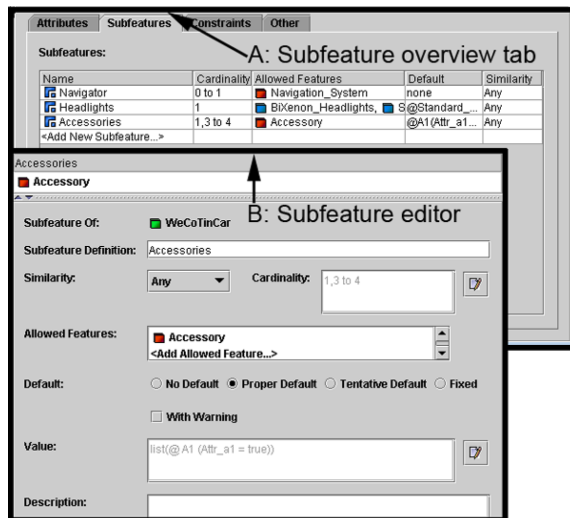

Fig. 5. Subfeatures of WeCoTinCar, and the subfeature editor for `Accessories`. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-2012-0547.)

the Modeling Tool, this editor (Area B) occupies the area of Feature type overview (Fig. 4, C).

3.3.6. *Constraints in PCML*

A feature type can define constraints. PCML has two types of constraints: soft and hard. *Hard constraints* define conditions that a correct configuration must satisfy. A configuration is considered consistent if and only if no hard constraint in any feature individual is violated. If any hard constraint is broken, the purchase process cannot be completed, e.g., placing the configuration to the shopping cart is prevented.

In contrast, a violated *soft constraint* issues a warning to the user, who can suppress or "silence" the warnings at will. However, depending on the modeler decision, the user may be required to review or even explicitly accept all warnings before proceeding in the acquisition process.

Constraints are expressed in a constraint language and named uniquely within a feature type. The first-level building blocks of the constraint language are *references* to access subfeatures, attributes of features, and constants such as integers or strings. References can be used in succession, e.g., to access an attribute of an individual realizing a subfeature. *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. Attribute references can be used with constants in arithmetic expressions that can be compared with the usual relational operators to create a test. Tests can be further combined into arbitrarily complex Boolean expressions using the standard Boolean connectives.

In the running example, constraints are defined on lines 37–46 in feature type `WeCoTinCar`, and on lines 52–54 in feature type `BiXenon_Headlights`. The latter uses a special variable `$config` to access the configuration individual and its attribute value.

3.3.7. *Constraints in the Modeling Tool*

The Modeling Tool provides several ways to define constraints: textually, graphically, and as table constraints. Tab `Constraints` (Fig. 5, B) in Feature type overview lists the constraints of the active feature type in textual form and facilitates adding new constraints. Next, graphical and table constraints in the Modeling Tool are discussed in detail.

Graphical constraints. Many existing constraints are relatively simple, often in the form of requires- and incompatible- relationships. To make entering and modifying simple constraints easier, a graphical constraint editor was developed [61]. It is possible to use the '`and`' and '`or`' – connectives to create sub-expressions. These sub-expressions can be used, e.g., as requiring, required, or incompatible factors in more complex graphical expressions.

Figure 6 illustrates graphically adding a hard constraint. First, the constraint is named, and the incompatible symbol is brought to the canvas; the resulting placeholders are shown in Fig. 6, A. Drag & Drop support frees the user from tedious and error-prone typing of object names and values. The placeholders are filled by dragging from the Feature tree. By default, dragging an allowed type creates a test for checking to ensure the subfeature is realized by an individual of the dragged type. Dragging an attribute value creates by default an equality test.

Accepting the defaults leads to the state of Fig. 6, B, and the following constraint:

```
constraint Sequential_transmission_
incompatible_with_Standard_headlights
not ( ( Headlights individual of
      Standard_headlights ) and
( Transmission = "Sequential_Manual" ) )
```

Table constraints. Often configuration and engineering information is represented in tables, where each row of the table corresponds to a viable combination of subfeature selections, parameter values, etc.

The Modeling Tool provides a user interface for easily constructing such tables. The example of Fig. 7

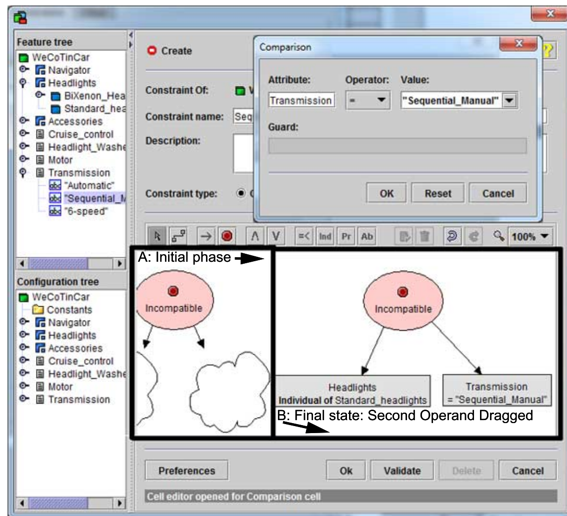

Fig. 6. Creating a graphical constraint (incompatible-with). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-2012-0547.)

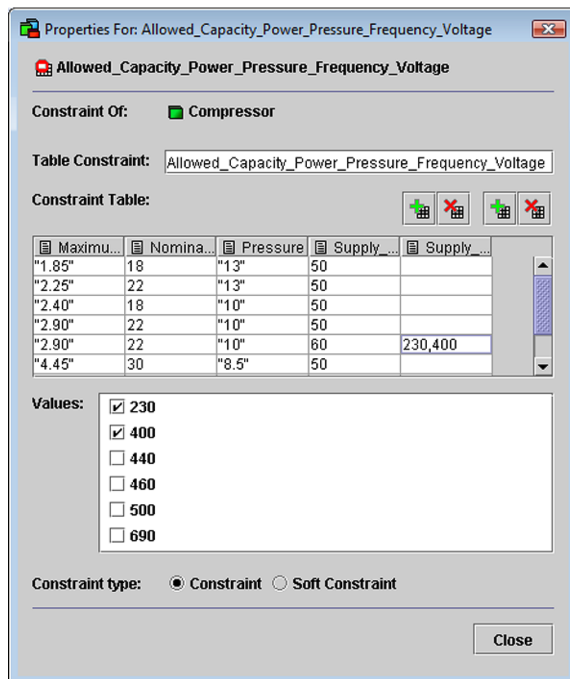

Fig. 7. A table constraint. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-2012-0547.)

comes from a real-world compressor family (several rows and extra space were removed). The fifth row (with the active cell) specifies that a 60 Hz 2.90 capacity compressor with nominal power of 22 and pressure 10 is available with voltages of 230 V and 400 V. Checkboxes enable including possible values of an attribute or allowed types of a subfeature quickly and without typing errors. Relevant attributes or subfeatures can be added to a table by browsing the configuration model. Additional flexibility is offered by the possibility to add textual sub-expressions.

3.3.8. Default values and realizations

In PCML, a default value is either *proper* or *tentative*. Both are considered when determining the validity of a feature individual, but only proper default values contribute to completeness. WeCoTin supports proper defaults with provisions for tentative defaults.

A default value can be reinforced with a soft constraint. If the subfeature realization or attribute assignment is changed from the default, the user receives a warning message.

WeCoTin has a mechanism for specifying different sets of default values that capture market-area specific defaults and "customer standards" specifying combinations of selections that have been agreed with a customer. A *default value package* (a.k.a. *pre-selection package*) [32] assigns a set of values to attributes and realizations to subfeatures. A default can be reinforced with a soft or even hard constraint – a corresponding automatically generated constraint will be broken if an attribute value or subfeature realization other than the default is assigned. A spreadsheet-like editor is provided to edit and compare several default value packages simultaneously.

Default values in WeCoTin do not have clear semantics. Proper defaults reduce the number of selections that a user must make. Often in practical models, optional elements are deselected by default or the cheapest alternative is set as the default.

3.4. User interface modeling and generation

3.4.1. Layouts

A web-based user interface for the end-user is generated without programming. The user makes selections on an HTML form; for example, see the right side of Fig. 3. The general idea is that each selectable attribute or subfeature of a feature individual being configured generates a *question*. The modeler creates a *layout* to define how questions related to a feature individual are divided into pages, the order of questions, and, optionally, the type of input control used for a question. An ordered list of questions to be shown on a single page is a *group*. By default, questions of a group are answered before consequences are deduced. The modeler can

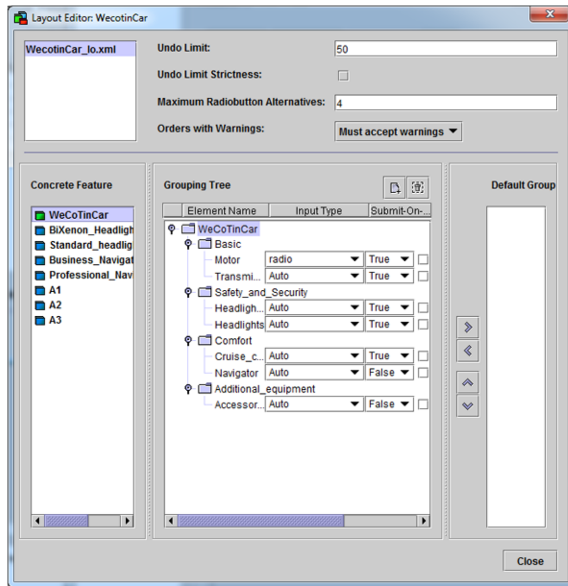

Fig. 8. Layout editor for Feature type WeCoTinCar. Group 'Basic' defines the Question area of Fig. 3. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/AIC-2012-0547.)

specify that inference is triggered immediately after a user responds to a question (`submit-on-change = true`). By default, all the questions are placed in a special group `default`. Questions can be hidden; helper variables can be applied without informing the user, for example.

The configurator automatically chooses a suitable input control type for each question (e.g., radio buttons, list box, checkbox). However, the modeler can override the selection, e.g., forcing radio buttons instead of a list box. When selecting one of several alternatives, `Maximum Radiobutton Alternatives` determines a threshold: how many alternatives are shown as radio buttons before switching to a list box. This can be overridden, as in the example of Fig. 8 for attribute `Motor` of `WeCoTinCar`. The user interface of the example in Fig. 3 reflects this layout.

Several layouts can be associated with a configuration model: to provide alternative views to a configuration model for different users, for summarizing a configuration, or for exporting to other systems. Order, grouping and visible objects can be varied as required.

3.4.2. Resources

In principle, the language of a configuration model is independent of the language(s) of the end-user interface. A *resource* file defines *display names* of objects of the configuration model to be shown to end-users and their optional *descriptions*. In addition, resources define *links to web pages* providing additional information, *messages for broken constraints*, etc. Multiple resources support multiple languages for end-users.

Configuration model objects can be named in a primary language, e.g., English. To help model maintenance, default display names are generated. For example, underscores in identifiers are replaced with spaces. Thus, creating a model with a suitable terminology facilitates near-automatic user interface generation in the primary language.

3.4.3. UI-models – Resource and layout bundles

A bundle of a resource file, a layout file, and a general user interface localization resource file comprises a *UI-model*. A configuration session is associated with three UI models: for configuring, for displaying a summary of the configuration and for exporting configurations. A separate UI model for exporting facilitates, e.g., a different language of an order specification to be sent into own organization. In our example integrations, see Section 3.7, export resources were used to map feature individuals to item codes (material numbers), and export layouts were used to specify a suitable ordering of exported items. Layouts, resource and UI-models are stored as XML files.

3.4.4. Other user interface aspects

Web-page templates define the general look of the user interface. They support maintainability of a configurator through separating the definition of visual appearance from the product-dependent parts of the user interface. Customized and shared templates provide the visual appearance of a company, and carry the visual appearance over to new products or product generations. Templates provide frames for dynamically generated contents (areas A, B, C and E of Fig. 3). HTML, ECMAScript [8] and Cascading Stylesheets (CSS) [3] are applied.

The user interface of the running example (Fig. 3) contains the WeCoTin logo and a static car picture as decorative visual elements.

3.5. Determining price and delivery time

WeCoTin has two mechanisms for determining prices. The *basic pricing mechanism* suits simple additive prices, while the more complex *advanced calculation mechanism* enables freely specifiable calculations.

The basic pricing mechanism is simple: each feature individual has the price specified by its type; the default is no price (0). In addition, each attribute value can be given a price. Prices of feature individuals and

prices of attribute values in a configuration are summed in the web browser after each selection. The user interface displays prices in the context of selecting a realizing feature type or an attribute value. For example, Fig. 3 shows prices of two attributes. Prices are specified in XML format price lists. Several price lists can be created, enabling support for several market areas, currencies, or special offers. A configuration session is associated with a single price list.

The advanced calculation mechanism [31] can perform calculations as a function of the current configuration, configuration model, and data in specific databases or XML files when contacting the Configurator Server. Typical uses are determining price, delivery time and possibly cost.

The mechanism processes *calculation items*, each equipped with a *condition expression* examining the configuration and a set of *assignments.* Assignments from calculation items whose condition expression evaluates to `true` provide values to *value sets.* Concrete values to assign can be retrieved from a database. Value sets are used as inputs to calculation functions that output other value sets. When calculating prices, the sum of items in a value set such as "`list_price`" is often calculated. Others functions include multiplication (e.g., to calculate discounts), subtraction, minimum and maximum. Furthermore, user-defined functions can be defined with Java. A calculation result (e.g., price) to be shown is selected from a template.

For determining delivery time, specific configuration decisions cause assignment of the whole configuration or a subsystem to a specific delivery time class, whose current lead times are maintained by, e.g., the production control function. The total delivery time is calculated, e.g., through maximum and sum operations.

3.6. Relationship to configuration ontology

The modeling concept 'feature' of PCML corresponds to the modeling concept 'function' of [44]. Functions and features are identical with respect to taxonomy, compositional structure and attributes. Neither features nor functions have ports or resources. However, PCML simplifies the conceptual model of [44] significantly. All subfeatures and feature types are considered exclusive – sharing of individuals as parts is thus not permitted. *Has-part inheritance definitions* are not supported – these special constraints allow the model to specify that certain properties of individuals of whole types are dependent on the properties of parts and *vice versa*. A straightforward example is a part that inherits the color of the whole, or an attribute value like width of a conveyor belt system being inherited by components. Further, all feature types are dependent, except the configuration type, which is independent. The configuration type has exactly one individual that serves as a root of the compositional structure. Because topological concepts and resources are excluded, contexts are excluded as redundant. WeCoTin does not support two-level configuration with separate sales configuration (features/functions) and technical configuration (components) views. Thus, implementation constraints are also excluded. Furthermore, constraint sets are not implemented.

3.7. Applying weight constraint rules and Smodels to provide inference

In this subsection, we describe how inference is provided for PCML configuration models. Three issues important for efficiency of the system are addressed in the implementation: off-line compilation of configuration models, limiting a configuration to a finite size in a semantically justified way, and symmetry breaking. First, Smodels and its language Weight Constraint Rule Language (*WCRL*) are summarized. These provide inference of WeCoTin and declarative semantics to PCML. We proceed by showing how PCML programs are compiled into WCRL, and then outline how WeCoTin uses Smodels to provide complete and sound computation of configurations satisfying requirements.

3.7.1. Smodels, WCRL and BCRL

Smodels is a system that follows the *answer set programming* paradigm, in which the problem is expressed as a theory consisting of logic program rules with clear declarative semantics, and the stable models of the theory correspond to the solutions (*answer sets*) to the problem [30]. The theories are expressed in WCRL. WCRL is equipped with *weight constraints* for representing weighted choices with lower and upper bounds and with conditional literals restricted by *domain predicates* to encode sets of atoms over which the choices are made [30]. By default, atoms have a weight of one. WeCoTin applies these default weights.

Smodels is a state-of-the-art implementation of Weight Constraint Rules. The Smodels system is based on a two-level architecture where, in the first phase, a front-end component, *lparse*, compiles a WCRL program with variables into programs in Basic Constraint

Rule Language (*BCRL*) that contain no variables. This provides a pre-compilation approach where some work is performed off-line. Lparse exploits efficient database techniques but does not resort to search.

The main functionality of the Smodels system is to compute a desired number of stable models for a BCRL program. Smodels allows a user to provide requirements (through so-called *compute statements*) to constrain the stable models to be computed. 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 is Davis–Putnam like, works in linear space, and employs efficient search-space pruning techniques and a powerful, dynamic, application-independent search heuristic. In addition, an efficiently computable approximation provides a set of atoms that must hold in any stable model if one exists, a set of atoms that cannot hold, and a set of unknown atoms. Smodels is implemented in C++ and offers APIs through which it can be directly integrated into other software.

3.7.2. *Configuration model and configuration representation overview*

Semantics of the PCML modeling language are provided by mapping configuration models expressed in PCML to Weight Constraint Rules. In addition, the mapping provides inference by making it possible to deploy Smodels as the configuration engine.

The configurator compiles a PCML program into WCRL, then into BCRL. The basic idea is to treat the sentences of the modeling language as shorthand notations for a set of sentences in WCRL. 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 of 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.

3.7.3. *Compilation process and WCRL program overview*

Compilation of PCML to WCRL follows the idea represented in [43]. The compilation process is summarized in Fig. 9. Here, *Configuration modeling core* in *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 *Inference Engine Interface* in Model manager translates the configuration model to a WCRL program. Data structures representing the PCML configuration model are saved for later use as serialized Java objects. The generated WCRL program includes sentences for:

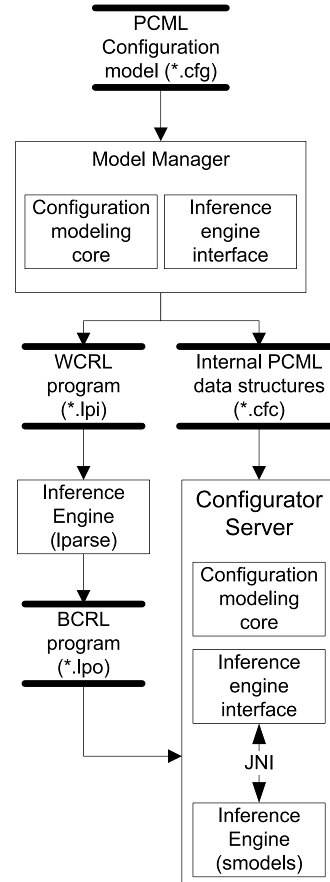

Fig. 9. Information flow of PCML compilation. First, the Model manager generates a WCRL program and serialized PCML data structures. WCRL is grounded to BCRL by lparse. BCRL and the serialized PCML data structures are used by the Configurator Server for repetitive configuration tasks.

- a set of standard axioms, called the ontological definitions,
- a set of sentences representing the configuration model, without constraints, including feature type hierarchy, compositional structure and attributes,
- a set of sentences representing the constraints in the configuration model,
- a set of ground facts representing the individuals out of which a configuration can be constructed,

and
- sentences for symmetry breaking.

Finally, the WCRL program is translated to BCRL using the 'lparse' component of the Smodels system.

In the following, these sentence categories are discussed, and the running example is used to give concrete examples. Table 1 summarizes the predicates resulting from the compilation.

Table 1
Predicates resulting from WCRL compilation

Predicate	Explanation
in (*c*)	True iff feature individual `c` is included in the configuration.
pa (*c1*, *t1, c2*, pn)	True iff feature individual `c2` is in the configuration realizing part `pn` of feature individual `c1` that is directly of type `t1`.
ppa (t, *c1*, *c2*, pn)	A domain predicate enumerating the possible part individuals for a part. Feature individual `c2` is a possible part for realizing `pn` of feature individual `c1` directly of type `t`.
pan(x)	`x` is a part name. Domain predicate `pan(x)` enumerates all part names.
compT_x(*c*)	Feature individual `c` is of the feature type `x`. `x` is replaced with the actual name of the feature type.
for (wt, *c1*, *c2*, pn)	Individual `c2` is allocated for use in part `pn` for feature individual `c1` of type `wt`. `for` is a symmetry breaking predicate, effective `ppa` atoms are generated through `for`-atoms.
prio (X, Y)	Individual `X` has priority over individual `Y`. In other words, `in(Y)` is possible only if `in(X)` is true. Used for symmetry breaking.
emptyPart (c, pn)	Helper predicate that is true iff feature individual `c` is in the configuration, and part {pn} has an empty realization.
prop_x (*c*, t, *v*)	True iff feature individual `c` of type `t` is in the configuration and has an assignment to attribute `X` with value `v`. `x` is replaced with the name of the attribute.
prSpec _n(v)	Domain predicate that enumerates possible value types for value type `n`. A new `n` is generated as new value types are encountered. Several attributes of the same value type can share the domain.
sat (t, *c*, cn)	A head for a constraint with the name `cn` in the feature individual `c` of type `t`, which is true iff the constraint `cn` is satisfied for individual `c`.
sc(t, *c*, cn)	A head for a soft constraint with the name `cn` in the feature individual `c` of type `t`, which is true iff the soft constraint `cn` is satisfied in individual `c`.
compTDom(c)	Domain predicate that enumerates all feature type names. `c` is a feature type.
isa(X, Y)	Feature type `X` is a subtype of feature type `Y`.

3.7.4. Ontological definitions

The set of ontological definitions remains the same in all configuration models. The following ontological definitions are created:

(1) Require that each feature individual in a configuration, except the root individual, is a part of another feature individual (line 001).
(2) Require exclusive parthood – a feature individual can be included as a part in at-most one feature individual with a given subfeature name (002).
(3) Specify transitivity of the is-a relation (004–005).
(4) Specify reflexivity of the is-a relation (006).
(5) Require that exactly one individual of the configuration type must be in the configuration (039).

```
001 in(C2) :- pa(C1,T,C2,Pn), ppa(T,C1,
C2,Pn).
002 :- 2{pa(C1,T,C2,Pn):ppa(T,C1,C2,
Pn)},
003 compT_Feature(C2).
004 isa(X,Z):- isa(X,Y), isa (Y,Z),
005 compTDom(X), compTDom(Y),
compTDom(Z).
006 isa(X,X):- compTDom(X).
039 1{in(C):compT_WeCoTinCar(C)}1.
```

3.7.5. Sentences representing feature types

Primarily, types are represented by unary domain predicates ranging over their individuals. Further, `isa(X,Y)` predicates are generated to represent the type hierarchy and to specify that individuals of a type are also individuals of its supertypes. In the example, lines 086–088 represent individuals of types `Professional_Navigation_System` and `Business_Navigation_System`. Further, `Feature`, `Navigation_System`, and `Business_Navigation_System` belong to the domain of feature types (`compTDom`) (007, 008, 032), `Navigation_System` is-a `Feature`, and `Business_Navigation_System` is-a `Navigation_System`. Feature individuals of these types are assigned also to their supertypes (033, 009). Generation of individuals is discussed in Section 3.6.9.

```
086 compT_Professional_Navigation_System
087 (ind_compT_Professional_Navigation_
System_1).
088 compT_Business_Navigation_System
088b (ind_compT_Business_Navigation_
System_1).
007 compTDom(compT_Feature).
```

```
008 compTDom(compT_Business_Navigation_
System).
032 compTDom(compT_Navigation_System).
034 isa(compT_Navigation_System,compT_
Feature).
010 isa( compT_Business_Navigation_
System,
011 compT_Navigation_System ).
033 compT_Feature(C) :-
034 compT_Navigation_System(C).
009 compT_Navigation_System(C):-
009b compT_Business_Navigation_
System(C).
```

3.7.6. Sentences representing subfeature definitions

For each subfeature definition, sentences are generated that allow the correct number of possible individuals as subfeature, determined by the cardinality. For example, subfeature (part) name `Accessories` of the running example (line 053) allows 1 to 4 individuals as part (line 054). All individuals have the implicit weight of 1. Possible individuals for realizing the subfeature are indicated with possible-part predicates (`ppa`) defined through symmetry breaking `for`-predicates (line 057), discussed in Section 3.6.10. If intermediate cardinalities are not allowed (such as 2 in the example) appropriate conflict sentences are generated (line 055). Further, individuals that are not possible parts are denied to be parts, line 056. In case of an optional subfeature (allowing cardinality 0) such as `Navigator` (lines 041–045), an `emptyPart` predicate is generated to make it easier to recognize and require (in a compute statement) an empty realization. For example, line 045 shows the `emptyPart` predicate of the `Navigator` subfeature.

Individuals that can realize each subfeature are generated separately, and they are allocated to each subfeature instance by symmetry breaking predicates discussed in Section 3.6.10.

```
053 pan(part_Accessories).
054 1{pa(C1,compT_WeCoTinCar,C2,
054b part_Accessories):
054c ppa(compT_WeCoTinCar,C1,C2,
054d part_Accessories)}4 :-
054e in(C1),compT_WeCoTinCar(C1).

055 :- 2{pa(C1,compT_WeCoTinCar,C2,
055b part_Accessories): ppa(compT_
WeCoTinCar,
055c C1,C2,part_Accessories)}2,
055d in(C1), compT_WeCoTinCar(C1).

056 :- compT_WeCoTinCar(C1), pa(C1,
056b compT_WeCoTinCar,C2, part_
Accessories),
056c ppa(T,C1,C2,part_Accessories),
056d not ppa(compT_WeCoTinCar,C1,C2,
056e part_Accessories).

057 ppa(compT_WeCoTinCar,C1,C2,
057b part_Accessories) :-
057c compT_WeCoTinCar(C1),compT_
Accessory(C2),
057d for(compT_WeCoTinCar,C1,C2,
057e part_Accessories).

041 pan(part_Navigator).

042 0{pa(C1,compT_
WeCoTinCar,C2,
042b part_Navigator) :ppa(compT_
WeCoTinCar,
042c C1,C2,part_Navigator)}1 :-
042d in(C1),compT_WeCoTinCar(C1).

043 :- compT_WeCoTinCar(C1),pa(C1,
043b compT_WeCoTinCar,C2,part_
Navigator),
043c ppa(T,C1,C2,part_Navigator), not
ppa(
043d compT_WeCoTinCar,C1,C2,part_
Navigator).

044 ppa(compT_WeCoTinCar,C1,C2,part_
Navigator)
044b :- compT_WeCoTinCar(C1),
044c compT_Navigation_System(C2), for(
044d compT_WeCoTinCar,C1,C2,part_
Navigator).

045 emptyPart(ind_compT_WeCoTinCar_1,
045b part_Navigator) :-
045c in(ind_compT_WeCoTinCar_1), not
045d pa(ind_compT_WeCoTinCar_1,
045e compT_WeCoTinCar,ind_compT_
045f Professional_Navigation_System_1,
045g part_Navigator), not
045h pa( ind_compT_WeCoTinCar_1,
045i compT_WeCoTinCar,ind_compT_
Business_
045j Navigation_System_1, part_
Navigator).
```

3.7.7. Sentences representing attributes

An attribute having a specific value in a feature individual is represented as a tertiary predicate specifying the feature individual, its direct type, and the attribute value. An attribute of a feature type is required to have

exactly 1 value from its domain (or 0 to 1 in case of an optional attribute) when an individual of the type is in a configuration. Line 121 specifies attribute `Motor` of `WeCoTinCar`. Possible attribute values are represented as object constants, and all possible values of an attribute are designated to a domain. Each possible value has the default weight 1. Domains can be shared among different attributes, and named domains generate similar domains. Possible values of obligatory attribute `Motor` of `WeCoTinCar` are specified in lines 122–126.

```
121 1{prop_WeCoTinCar_Motor(X,compT_
WeCoTinCar,
121b Y): prSpec_5(Y)}1:- in(X),
121c compT_WeCoTinCar(X).
122 prSpec_5("20i").
123 prSpec_5("25i").
124 prSpec_5("25d").
125 prSpec_5("30i").
126 prSpec_5("30d").
```

3.7.8. Constraints

Constraints expressed in PCML are translated to equivalent constraints in WCRL in a manner that ensures constraints defined at type level apply separately to each individual of the type. A constraint is only required to apply when the individual bound to it is included in the configuration. The translated constraints do not provide groundedness to individual atoms. Additional details are beyond the scope of this paper.

3.7.9. Facts for representing individuals

We took the approach that the set of individuals out of which the configuration can be constructed is predefined. 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: The configuration individual serves as the root of the compositional structure. Because a maximum cardinality is defined for every subfeature definition, we can calculate an upper bound of the number of needed individuals. First, the configuration individual is generated. For each subfeature definition of the configuration type, the number of individuals defined by the maximum cardinality of the subfeature definition is generated for each allowed concrete subfeature (part) type. This is performed recursively to generate subfeature individuals for all subfeature 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 compositional 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 based on constraints or mutually exclusive branches of the compositional structure. For example, a subfeature definition with maximum cardinality of N with M allowed types generates N*M individuals, although at maximum N can ever be used.

The available individuals are represented as object constants with unique names. Two individuals (`ind_compT_ A1_1` and `ind_compT_A1_2`) of type `A1` for part `Accessories` of the running example are shown below. They are listed with `for`-predicates used for symmetry breaking, discussed in the next subsection. Individuals `ind_compT_ A1_3` and `ind_compT_A1_4` are defined similarly.

```
069 compT_A1(ind_compT_A1_1).
069b for(compT_WeCoTinCar,
069c ind_compT_WeCoTinCar_1,ind_compT_
A1_1,
069d part_Accessories).

070 compT_A1(ind_compT_A1_2).
070b for(compT_WeCoTinCar,
070c ind_compT_WeCoTinCar_1,
070d ind_compT_A1_2,part_Accessories).
```

3.7.10. Symmetry breaking sentences

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. Next, we describe two forms of unwanted symmetries and present a method used by WeCoTin to break them.

The first form of symmetry arises when several individuals directly of a type are possible parts with a subfeature name for an individual. For example, in Fig. 10(a) type `A` has subfeature 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`. They can both realize subfeature (part) `P` of individual `a-1`. More precisely, individuals `b-1` and `b-2` can be a subfeature with subfeature name `P` in `a-1`. The configurations in Fig. 10(b) and (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 subfeature individuals directly of the same type are always used in a fixed order. The individuals are ordered by giving them priority rankings. A lower

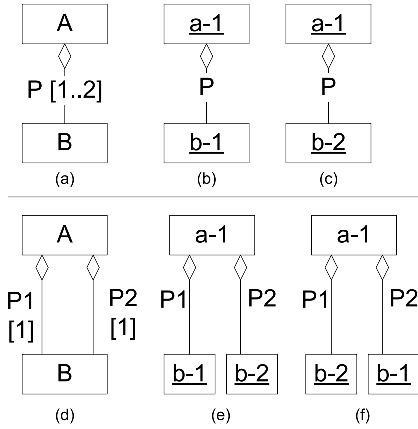

Fig. 10. Symmetry breaking in WeCoTin.

priority individual is not allowed in the configuration if all the higher priority individuals are not included in the configuration. Symmetry breaking would thus allow only the configuration in Fig. 10(b).

The second form of symmetry is shown in Fig. 10(d)–(f). Without symmetry breaking, any individual of type `B` could realize any subfeature (part) where type `B` is as an allowed type. For example, individuals `b-1` and `b-2` of type `B` could both realize either `P1` or `P2` in individual `a-1`. We break this form of symmetry by allocating each part individual to a specific whole individual and subfeature name. After allocation, either (e) or (f) is allowed.

Lines 69–70 (and 71–72, not shown) above showed facts representing individuals of `Accessory`-type `A1`, and their allocation via `for`-predicates. Priorities of individuals of type `A1` are defined below (lines 79–81). Denial to use lower priority individuals before higher priority ones is specified on line 89. Transitivity of priority is specified on line 90.

```
079 prio(ind_compT_A1_1,ind_compT_A1_2).
080 prio(ind_compT_A1_2,ind_compT_A1_3).
081 prio(ind_compT_A1_3,ind_compT_A1_4).
089 :- prio(X,Y), compT_A1(X),
compT_A1(Y),
089b in(Y), not in(X).
090 prio(X,Z) :- compT_A1(X),
compT_A1(Y),
090b compT_A1(Z),prio(X,Y), prio(Y,Z).
```

3.7.11. Overview of inference

The configuration engine smodels uses the compiled BCRL form of the configuration model. At the beginning of a configuration session, the Configurator Server loads the data structures representing the configuration model into the configuration modeling core and the corresponding BCRL program into smodels. See Fig. 9.

A compute statement representing current requirements is set through the smodels API; a compute statement requires a relevant set of atoms to be true and another set of atoms to be false in an answer set representing the configuration. Consistency of the requirements is checked by constructing a suitable configuration, enabling early detection of a possible dead end.

Deducing the consequences of requirements is based on computing an efficient approximation (*well-founded approximation*) of the set of configurations satisfying the requirements. Intuitively, the approximation contains a set of atoms that must be true in a configuration satisfying the requirements, a set of atoms that cannot be true, and a set of unknown atoms [30].

There are several "levels" of requirements that the configurator uses: user-confirmed selections are the lowest level (i.e., the highest priority) and they will always be respected; user-confirmed selections augmented with defaults enforced with soft constraints form the next level; finally, the most comprehensive requirements include additionally proper defaults. A maximum level of requirements where a configuration can be found is determined. On this level, consequences are deduced to support the user. With the aid of the well-founded approximation, most consequences of current requirements are detected. For example, the only compatible remaining alternative is selected automatically. If this results in new consequences, they are taken into account as well, due to fixpoint semantics of the approximation. Finally, if defaults have been removed, an attempt is made to restore defaults that are not in direct conflict of deduced consequences. If successful, a configuration including those defaults is returned. Otherwise, defaults remain discarded until the next contact with the Configurator Server.

Based on this approximation, the Smodels interface generates a new configuration and hands it to Configurator Server. Configurator Server uses its calculation subsystem to compute results such as price or delivery time before the configuration is returned to WebUI servlet, which shows it to the user.

WeCoTin uses significant effort to "gray out" incompatible alternatives. The idea is simple: assume that a question that will be shown next has no answer, and set other previous answers as requirements. Atoms related to the question that are on the list of necessarily false atoms of the well-founded approximation are not

compatible with current user inputs and will be grayed out. This process is repeated for all questions that will be visible on the next set of questions. Haag [15] documented the same process for determining graying information for next visible questions (calculating "Dynamic User Domains"). An explanation for graying is available by selecting the grayed out selection – some constraint(s) will be broken, and corresponding messages explain the reason.

3.8. Set-up of configurator and configuration model

The process to set-up a configurator does not require programming. On an installed system set-up, one or more configuration models and corresponding layouts and resources are created and deployed. In addition, a web page or site providing a link to invoke the WebUI servlet (see Fig. 2) with the desired configuration model and other configuration session invocation parameters is required. The parameters include UI models for configuring, showing a summary, and exporting the finished configuration. Further, a price list for the basic price mechanism and an *exporter* (see the next paragraph) are specified. Optionally, it is possible to specify a configuration to load and a pre-selection package to use, and to define the calculations of the advanced calculation mechanism (Section 3.5).

An *exporter* (implementable in JAVA) delivers a configuration to external system(s); an API provides access to relevant information such as the configuration, UI models, and the configuration model. A default exporter creates an XML representation of the configuration and stores the configuration summary page. A tailored exporter can transfer a finished configuration to an ERP, PDM, or e-commerce system. Implemented exporters include Intershop 4 e-commerce system, EDMS2 product data management system, and Vertex G4 3D CAD. The CAD exporter generates a 3D visualization during the configuration process.

The remaining activities for creating a configurator include customizing the templates and style sheets to give the desired company look. A full e-commerce system requires functionalities outside the scope of WeCoTin, including customer account management (e.g., volume and customer dependent discounts, integration of personal saved configurations), buying and order tracking functionality, and ordering of non-configurable items such as spare parts.

4. Evaluation: Characterization of practical cases, modeling and configuration support, and performance

According to the third guideline of Design Science Research [19], the utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. IT artifacts can be evaluated in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization and other relevant quality attributes. The scope of this work allows only an evaluation of a subset of these criteria. The views chosen for evaluation are high-level efficacy and performance.

Configuration models, related configuration tasks, and their IT support (configurators) could be characterized from several perspectives. These include the size of the models, computational performance and complexity, and effectiveness of specific modeling techniques related to specific configuration tasks as defined by the company offering. Less technical views cover aspects of practical interest such as the proportion of cases covered by the configuration models, completeness of the models in terms of business requirements, usability, and different aspects of utility provided and sacrifices required.

High-level efficacy of the main artifact WeCoTin was demonstrated by showing that practical configuration problems could be solved [53]. 26 configuration models were created to evaluate and demonstrate the applicability of WeCoTin and PCML to real-world industrial configuration problems, which demonstrates the high-level efficacy of the constructs. The models were characterized with numerous indicators of size and degree of application of different modeling constructs. All modeling was performed by researchers who developed this system. Due to limitations of space, in this paper we show only the main characterizations and provide extended conclusions.

Empirical performance evaluation of WeCoTin with four real-world products has been conducted, as originally presented in [59]. Evaluating configurator performance is important because of at least NP complexity class of configuration problems [41]. To quantify the performance challenge, a method was developed that allows run-time performance testing of configurators based on real-world products and random requirements. We summarize the method and key results.

4.1. Characterization of modeled cases

WeCoTin configurator can model and support configuration tasks of several industrial domains. This was verified by modeling the sales view of 14 real-world products in their entirety (some with extra demonstration features, one in 2 variants) and 8 partial products or concepts. Configuration sessions were performed with each model. In this subsection, configuration models created with WeCoTin will be characterized. The Modeling Tool was instrumented to provide the characterizing metrics based on static configuration model analysis. The configuration models come from the following domains:

- Machine industry: 8 models from 3 companies; 6 compressors (1 twice), 1 undisclosed vehicle and 1 military vehicle.
- Healthcare: 3 models from 2 companies; 2 dentist's equipment and 1 hospital bed.
- Telecommunications: 4 models from 2 companies; 3 mobile and 1 broadband subscription.
- Insurance services: 3 models from 1 company; insurance products.
- Maintenance services: 2 models from 1 company; 2 generations of elevator maintenance contracts.
- Software configuration: 1 model; Debian Linux with package versions.
- Construction: 1 model; modular fireplace.
- Demonstration models: 3 cars, 2 based on a subset of a real-world car and 1 fictional.

This subsection is structured as follows. Section 4.1.1 gives an overview of the configuration models, their background, and validation. Section 4.1.2 describes component type hierarchy, overall configuration model size, and price modeling. Section 4.1.3 details the compositional structure of the models, Section 4.1.4 describes attributes, and Section 4.1.5 identifies constraints.

4.1.1. Model background, status and validation

In most cases, material from case companies, such as order forms, brochures, and other documentation, were used as a basis for configuration modeling. Company experts were often contacted for additional information. In some cases, we re-engineered configuration model information from company websites.

Five models from three companies were test-used by company representatives. In addition, configuration demonstrations and immediately-following focus groups were used to validate an additional seven configuration models. One model (23, Linux) was significantly larger than others and semi-automatically generated. Discussions on model characterizations exclude this model, but averages and totals were calculated with and without it (see Tables 2 and 3).

Table 2

Main characterizations of the configuration models. TT = Total number of Types, AT = Total number of Abstract types, CT = Total number of Concrete types, ST = Total number of types as Subtypes, %S = % of types as Subtypes, QU = total number of QUestions, %R = % of questions in Root of compositional hierarchy

Model	TT	AT	CTs	ST	%S	QU	%R
1 C FM	9	2	7	4	44	31	58
2 CFm sc	9	2	7	4	44	31	58
3 C FS	3	0	3	0	0	24	88
4 C FX	1	0	1	0	0	20	100
5 C FL	9	2	7	4	44	28	64
6 C M	3	0	3	0	0	23	91
7 KO old	5	0	5	0	0	28	79
8 KO new	15	3	12	7	47	77	4
9 Bed	31	8	23	27	87	34	76
10 Firepl	7	1	6	4	57	4	75
11 Pasi	5	1	4	2	40	79	95
12 Dental	64	11	53	43	67	109	3
13 X-ray	11	2	9	4	36	37	41
14 Vehicle	28	4	24	9	32	24	75
15 Ins 1	8	2	6	5	63	30	20
16 Ins 2	62	13	49	56	90	49	20
17 Ins 3	11	3	8	5	45	41	29
18 Ins 4	37	11	26	34	92	242	5
19 Mob 1	4	0	4	0	0	18	56
20 Mob 2	39	9	30	38	97	65	25
21 Mob 3	5	1	4	3	60	21	38
22 Broad	66	15	51	64	97	485	1
23 Linux	626	1	625	624	100	4369	14
24 Iced	8	2	6	5	63	4	75
25 Wcar	6	1	5	2	33	10	60
26 CarDis	10	2	8	5	50	12	58
Total	1082	96	986	949		5985	
Total no Linux	456	95	361	325		1526	
Average	18	4	14	13	48	227	50
Avg no Linux	24	5	19	18	59	61	52
Median	9	2	7	5	46	31	58
Min	1	0	1	0	0	4	1
Max	626	15	625	624	100	4369	100

4.1.2. Taxonomy, model size and pricing

Numbers of abstract (AT), concrete (CT) and *total feature types (TT)* contribute to the size of a configuration model and are shown in the corresponding columns of Table 2. The total number of feature types

Table 3

Main characterizations of the configuration models. CO = total number of constraints, PR = PRice calculation mechanism, EA = total number of Effective Attributes, %IA = % of Inherited Attributes, ES = total number of Effective Subfeatures, %IS = % of Inherited Subfeatures

Model	CO	PR	EA	%IA	ES	%IS
1 C FM	17	adv	27	22	4	50
2 CFm sc	17	adv	27	22	4	50
3 C FS	14	adv	23	0	1	0
4 C FX	23	adv	20	0	0	–
5 C FL	13	no	24	17	4	50
6 C M	14	no	22	0	1	0
7 KO old	13	no	26	0	2	0
8 KO new	1	no	58	81	19	47
9 Bed	10	basic	31	0	3	0
10 Firepl	0	no	2	0	2	0
11 Pasi	13	no	77	3	2	0
12 Dental	36	no	76	70	33	79
13 X-ray	3	no	32	44	5	40
14 Vehicle	7	basic	8	0	16	0
15 Ins 1	4	no	20	10	10	0
16 Ins 2	0	no	19	58	30	27
17 Ins 3	14	no	29	0	12	0
18 Ins 4	84	no	189	26	53	51
19 Mob 1	6	basic	15	0	3	0
20 Mob 2	28	basic	52	29	13	0
21 Mob 3	6	no	20	60	1	0
22 Broad	43	no	453	89	32	6
23 Linux	2380	no	3745	67	624	0
24 Iced	3	basic	2	0	2	0
25 Wcar	3	basic	8	25	2	0
26 CarDis	3	basic	9	22	3	0
Total	2755		5014		881	
Total no Linux	375		1269		257	
Average	106		193	25	34	16
Avg no Linux	15		51	23	10	17
Median	13		25	19	4	0
Min	0		2	0	0	0
Max	2380		3745	89	624	79

varied from 1 to 626, the median was 9 and the average was 24. The *number of direct subtypes of abstract types* (excluding root of the feature type hierarchy `Feature`) (ST) characterizes application of the taxonomical hierarchy. As a percentage "%S" the figure varied from 0% to 100%, with the average without Linux being 59% and the median being 46%.

A selectable attribute or subfeature of a feature individual being configured generates a *question* during a configuration process. The *number of questions in a configuration model* ("QU") roughly characterizes the size of each configuration model and the related configuration task. In a typical configuration model without redundant concrete feature types, each question might need to be answered while configuring a product. All possible questions may not be asked during a configuration session, because an individual of a specific type is not necessarily selected into a configuration, or if some attributes or parts are defined to be invisible to the user or to have a fixed value. However, if several individuals of a feature type are included in a configuration, the number of questions may be multiplied. The average was 61 questions per configuration model and 5.4 questions per concrete type.

Especially simpler models focused on the configuration type, the root of the compositional structure. The degree of such concentration is characterized by the *proportion of questions defined in the configuration type* ("%R"). On average about half (50%), and median 58%, of questions were in the root feature type, with a large scale of variation. The average percentage of subfeature definitions in the configuration type was 70%. In 12 models, all subfeatures were defined in the configuration type. On average 46% of effective attributes were defined in the configuration type.

Column ("PR") in Table 3 specifies the applied price calculation mechanism. The "basic" price calculation mechanism (Section 3.5) with only additive prices was applied to four real-world products and three demonstration models. The "advanced" calculation mechanism [31] was used to determine the price for three products. Prices were often omitted either due to indicated sensitivity or to constrain modeler resource usage. The basic price calculation mechanism would have been sufficient for products other than compressors and insurance products.

4.1.3. Compositional structure

An indication of the use of compositional structure is given by the *number of effective* (inherited and locally defined) *subfeatures* in concrete types ("ES"), see Table 3. The average was 10 and the median was 4 for effective subfeatures (parts) per configuration model. The average *number of effective subfeatures per concrete feature type* was 0.7, which indicates relatively moderate usage of the compositional structure.

The compositional structure was mainly defined through subfeature definitions in concrete types; inheritance was used less frequently: 8 models (31%) applied inheritance of subfeatures whereas at least one subfeature was defined in 25 models (96%). The percentage of inherited subfeatures of all effective subfea-

tures is shown in column ("%IS"). On average, a model had 7 subfeature definitions in concrete types and one in abstract types. However, 4 of the effective 10 subfeature definitions in concrete types were inherited. Of these, 3 were applied as such and one was refined, e.g., to restrict the set of allowed types. Some models applied inheritance in compositional structure significantly more. For example, in a dentist's equipment configuration model, 79% of effective subfeature definitions were inherited (26 of 33) and 6 were refined. In the 8 models applying subfeature inheritance, 31% of effective subfeature definitions were inherited.

On average, 6 subfeature definitions were optional, that is, with cardinality 0–1, and 2 subfeature definitions were obligatory with cardinality 1–1. Only one demonstration model (the running example of this paper) contained a subfeature definition with a larger maximum cardinality than one.

The number of allowed types in subfeature definitions characterizes available variation of the compositional structure. On average, an effective subfeature definition had two effective allowed feature types. This relatively low number is partially explained by optional subfeatures with only one effective allowed type. The maximum number of effective allowed types of a model was on average 4 types, and maximally 15.

4.1.4. Attributes

All 26 models defined attributes. Attributes were the main mechanism for modeling variability; on average 83% of questions originated from attributes and the remaining 17% from parts. The *number of effective attributes* of the configuration model ("EA") is the sum of inherited and locally defined attributes in concrete types. The average was 51 effective attributes per model and 3.5 per concrete feature type.

Applying attribute inheritance can be quantified as follows. 16 (62%) models applied inheritance of attributes. Of the 1269 effective attributes, 51% originated from local *attribute definitions in concrete types*. The remaining 49% of effective attributes were inherited, see column ("%IA") in Table 3. 122 *attribute definitions in abstract types* were inherited as such into 537 attributes in subtypes, and into 168 attributes in refined form, for a total of 705 inherited attributes.[1] On average, a model contained 26 attribute definitions in concrete types. The average 5 *attribute definitions in abstract types* expanded to an average of 25 effective attributes in concrete types. The average *percentage of inherited attributes* in a concrete type was 23%. Some models applied attribute inheritance more significantly, e.g., 44–89% of effective attributes were inherited in some larger models.

Attribute value types were distributed as follows: 56% Boolean, 29% enumerated string, 9% integer, and 6% unconstrained string attribute definitions. Unconstrained strings specified additional details such as customer names, addresses, and other aspects that do not require inference.

Attribute domain sizes remained quite small. The most common domain size was 2–3 possible values in 82% of 717 attribute definitions with a fixed domain. 14% of attribute domains were of size 4–10. About 4% of attribute definitions had a domain size of at least 11 possible values. 1% of domains were of size 1, mostly created through attribute value refinements. The *maximum domain size* of a model varied significantly – the largest domain was 436 possible values, with an average of 41 and a median of 11 possible values.

One attribute per model (20 m 3%) was defined as *optional*; it is possible to specify in a complete configuration that no value will be assigned to the attribute.

4.1.5. Constraints

The number of *constraints*, column ("CO") in Table 3, varied from 0 to 84 (up to 2380 with Linux), an average of 15 and a median of 13 per model. On average, 2 of the constraints were soft, and the rest were hard. Inheritance of constraints was applied to some extent, because abstract feature types defined 44 constraints. Of these, 40 were hard and 4 soft.

It is not trivial to characterize complexity of constraints. We apply a simple syntactic metric: parse tree size of the constraint expression. For example:

```
Active_Cruise_Control_Requires_BiXenon
( Cruise_control = true ) implies
($config.Headlights individual of
BiXenon)
```

The "complexity" of the example constraint is seven. Complexity of a literal, a constant, a variable, a feature type or attribute reference, or other basic expression building blocks is one unit. Each operator application counts as one unit in addition to argument complexity.

Typical constraints were small. Almost half (45%) of the constraints were of roughly the same complexity (6–10) as the example constraint, and 36% were a bit more complex (11–20).

- 12% of constraints had complexity 0–5.
- 45% of constraints had complexity 6–10.
- 36% of constraints had complexity 11–20.

[1] 705 inherited attributes include 78 (= 705 − 627) attributes inherited to abstract types.

- 4% of constraints had complexity 21–50.
- 1% of constraints had complexity 51–100.
- 1% of constraints had complexity 101–1000.
- 1% of constraints had complexity over 1000.

Maximum constraint complexity varied significantly. The median was as low as 13 and the average without Linux was 235. The maximum complexity was 1319. All the compressor models had a large table constraint specifying feasible combinations of values of 5 attributes, each with a relatively large number of rows, which explains the unexpectedly high average.

4.2. *Evaluation of modeling and configuration task support*

This subsection contains evaluation of available modeling mechanisms, the Modeling Tool and Configuration Tool. First modeling efficacy is discussed, followed by estimates of modeling and potential integration effort. Next, modeling capabilities are discussed in relation to those encountered in our cases. Finally, the WeCoTin Configuration Tool is discussed from the end-user's point of view.

4.2.1. *Evaluation of modeling mechanisms*

Capabilities of PCML or WeCoTin did not limit the scope of modeling, despite the relatively small subset of modeling concepts available. In other words, the subset of modeling concepts of PCML was adequate for modeling the case products.

Attributes were the main mechanism for capturing the variability of the modeled products. Application of the compositional structure was important but perhaps less frequent than anticipated. A partial explanation is that often alternative or optional features were modeled as enumeration or Boolean attributes if configuring details of the selected feature was not needed. This saves the creation of feature types corresponding to the alternatives and joining them as allowed types.

Subfeature (part) definitions with cardinality were useful and convenient for modeling either an optional or alternative realization of a role in the compositional structure. This prevents the need for a number of extra constraints. For example, some commercial systems require that each alternative is specified as optional, and a mutual exclusivity constraint is required [6]. However, a minor inconvenience was apparent in the case of an optional subfeature (cardinality 0–1), and exactly one allowed type: it was difficult to invent a name for the feature type and for the subfeature (e.g., an optional radio would be described with a subfeature definition `radio` and a feature type `radio`). A bit surprisingly, large cardinalities were not needed.

Inheritance significantly saved modeling effort necessary for larger models. Almost half (49%) of effective attributes were inherited, and one definition in a supertype created on average 4.4 of effective attributes. Refinement was useful for limiting the domain of allowed values or allowed types in subtypes. Inheritance of the compositional structure was also useful, although it was applied only in about 31% of the models because the compositional structure was shallow and often concentrated on the configuration type. In larger models, about half of effective subfeatures were inherited.

Floating or fixed-point numbers or integers with very large domains would have been useful in the insurance and compressor domains. The domains allowed, e.g., free specification of monetary insurance coverage, or the calculated capacity of a compressor.

Neither explicit resource modeling nor topological modeling, e.g., ports [55], were needed. However, when modeling services and their delivery processes [55], there was a need to assign different stakeholders as resources that participate in different service activities. This assignment can be somewhat clumsily modeled with attributes. However, allocation of responsibilities to different, dynamically defined stakeholders could be more naturally modeled as connections between the activities and stakeholders.

4.2.2. *Evaluation of the Modeling Tool*

The researchers creating configuration models with the Modeling Tool considered the editing facilities generally very adequate. The visual type and part structure hierarchies, fluent attribute domain editing, and the graphical constraint editor all facilitated efficient modeling. After the graphical constraint editor became available, most constraints were created and edited with it. Global renaming support of objects without the need for text-based search-and-replace was convenient and likely to reduce potential for errors. Table constraints were very useful in the cases where they were applied.

User interface development was rapid due to dynamic generation of product dependent parts. Drag & Drop layout definition, and resources with automatically generated default display names were effective.

Some useful basic functionalities were not implemented, which created extra effort and potential sources of error. Most prominently, lack of inheritance of layouts and resources caused, e.g., the need to enter attribute display names or order of questions several

times. There was no editor for price lists and calculations, and related XML-editing was considered error-prone and not very convenient.

In some cases, it would have been useful to include questions related to several feature individuals to a single end-user interface question form. In some contexts, it would also have been useful to be able to hide some alternatives dynamically instead of graying them.

4.2.3. Modeling and integration effort

No reliable statistics on the total effort expended to create configuration models was gathered, but some estimates (subject to author bias) are provided.

WeCoTin lacks an integrated wizard for creating all aspects of a new empty configuration model and its deployment into a web-based application. Therefore, setting up a new model for a new company would require an experienced person (someone who occasionally creates these models) to spend roughly an hour or even two. Adapting templates to provide a distinct company look would necessitate similar effort.

The amount of work required to create a configuration model was dominated by knowledge acquisition and validation. Initial versions representing configuration options and rules of a well-documented order form (an A4 or so) can be performed in a matter of hours. A working configurator demonstration with a company look can be provided in a day, if based on readily available information. In the other extreme, the most often edited model (`1 Compr FM`) was checked in 24 times in our revision control system, and it was edited by six different researchers. The total effort was several person-weeks.

Two custom exporters were created for the compressors – one to export configurations to Intershop e-commerce system, and one to export them to EDMS2 product data management system. The EDMS 2 exporter took an experienced persona a few person-days to design, implement, and demonstrate with sample data. The Intershop integration and demonstration was implemented in about three person-months by a summer trainee who was neither familiar with Intershop nor any configurator. The effort included system installation, becoming familiar with the e-commerce system and WeCoTin, the design of the integration principle, implementation, and setting up a demonstration site with the visual look of the company.

4.2.4. Configuration Tool – End-user view

WeCoTin has not been in production use. However, a number of experiences have been gained in test-use by company representatives in three cases, and through numerous demonstrations.

The general reception of WeCoTin and its usability has been positive in companies where test-use has been arranged or where demonstrations have been performed. In one company, the Configuration Tool was considered better than the commercial product they applied. In the second company, there was a desire to model all products and plans to deploy the configurator to production use. These plans were ceased due to non-technical reasons.

The wizard-style user interface is relatively intuitive. The multi-step undo functionality is very useful, and error messages generated directly from PCML (with descriptions in resources) are often sufficient.

However, significant room for improvement exists. Active support for restoring the consistency of a configuration should be offered, e.g., with better explanatory facilities or through diagnoses and active fix proposals for an inconsistent configuration. These could be provided, e.g., by model-based diagnosis techniques [9,11]. Requirements are specified by direct selection of an attribute value or a feature type to realize a subfeature. It is not possible to specify a set (or a range) of satisfactory attribute values, or to select a supertype for later specialization. The possibility of navigating through the configuration tree may not be obvious, and the configuration tree of a large model may not fit on one screen. Further, automatic completion of configurations is not provided, although defaults typically provide a satisfactory form of automatic completion.

4.3. Empirical evaluation of performance

In [59], a method for empirical performance testing of configurators was described along with its application to four real-world products described above. This subsection provides a summary and adds some new results. First, the need for performance testing is briefly described and insight of the basic idea of the test method is given. Next, the test setup and main results are presented. Finally, a number of additional performance indications are offered. Discussion is presented at the end of this paper.

4.3.1. Motivation and potential approaches

Performance testing of configurators is important, as the configuration task is at least NP-hard in most formalisms, including the one in this work [21,41–43,49]. However, conventional wisdom in the configuration community is that solving typical configuration problems is relatively easy and does not exhibit this kind of exponential behavior. There are some documented

results on the efficiency of configurators [22,25,46, 49], but systematic and wide range empirical testing of configurators on real-world products that would show whether the wisdom is, indeed, wisdom, is still lacking.

A method was developed that allows performance testing of configurators. 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.

In principle, one could test configurator performance by *using real-world configuration models* or *randomly generated configuration* models. Random configuration model generation could be synthetic or use real-world products as a seed. Another dimension is the selection between *fixed* or *randomly generated requirements*. Our method is based on real-world configuration models with random requirements. As a motivation for using real-world configuration models, there is a risk that random models without a large set of real-world 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-world problems. Knowledge acquisition and modeling for a sufficient and justified seed of real-world models for random model generation would be a major task.

4.3.2. Test method: Simulate naïve user requiring attribute values or specific subfeature realizations

The idea of the test method is simulating a naïve user inputting random requirements when configuring a real-world product. For generating random sets of requirements, we consider how the configuration model appears to the user. There are menus (possibly multi-choice), radio buttons, and check boxes allowing the user to select between different alternatives. These represent specific attribute values or selection(s) of allowed type(s) to realize a subfeature. Guided by 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 the set of potential requirements that a user could state related to an attribute or a subfeature of an individual. With respect to the running example of Listing 1, a group could represent the selection of a value for the `Cruise_control` attribute or the selection of a type to realize subfeature `Navigation_System` in an individual of type `WeCoTinCar`. Each group has a number of *requirement items*, each representing a potential requirement, that is, a possible selection (an attribute value, an individual of allowed type, or selection of no alternative (`none`) if the selection is optional). The number of requirements that can be generated from a group is defined by *minimum* and *maximum cardinality*; both are one in case of an attribute definition. A group representing a subfeature definition has the maximum cardinality of the subfeature definition. The minimum cardinality is the maximum of one and the minimum cardinality of the subfeature definition. To sum up, requirement groups and requirement items represent all individual requirements one could state related to a configuration model by direct selections of possible answers, including deselecting optional elements.

A *test case* contains a specific number of requirement items. When generating a test case, a group is randomly selected to generate the number of requirements specified by the minimum cardinality. A requirement is generated from a group representing an attribute by choosing randomly one requirement item. Generating a requirement for a subfeature is slightly more complex. In WeCoTin, 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 allowed type (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. A group can be selected again to generate a new requirement, if the maximum cardinality allows. Group selection is repeated until the desired number of requirements has been generated.

4.3.3. Test setup

Performance tests were performed with 4 first-modeled real-world products. The models were 1 Compr FM, 3 Compr FS, 4 Compr FX and 14 Ve-

hicle, coming from two domains and characterized above.

We generated with a Java-based test generator random test cases that each contained a number of requirements. Each requirement specified a value for an attribute (including `none`, if allowed), or realization of a part with a specific individual (or no realization, if cardinality includes 0). 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 questions in each configuration model (see Table 2). 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.

The test setup is illustrated in Fig. 11. The tests were run on a laptop PC with a 1 GHz Mobile Pentium III processor, 512 MB RAM, and Windows 2000 Professional. A WCRL program was generated off-line for each PCML configuration model using the normal model compilation facilities of WeCoTin. A Java-based test driver executed each test case in sequence. 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 compute statement contained as a requirement an atom corresponding to each requirement item 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. The test driver captured and analyzed smodels output and performed timings. If a configuration was found with the requirements of the test case, the test case was considered *satisfiable*; otherwise, it was considered *unsatisfiable*.

4.3.4. Test 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.

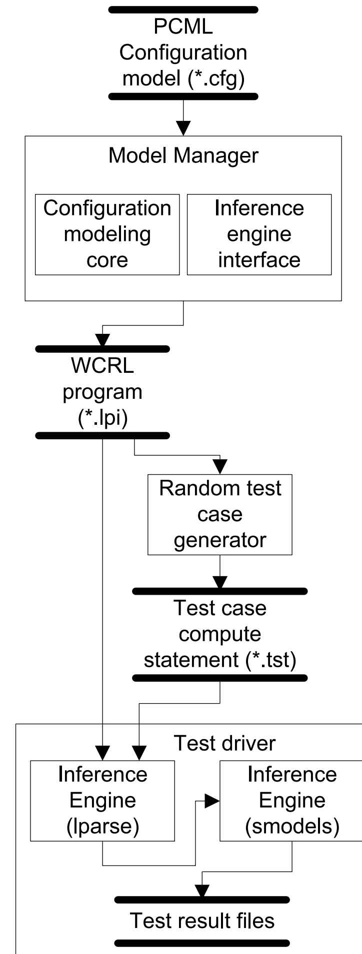

Fig. 11. Test setup. A PCML configuration model is compiled into WCRL off-line. A test case and the compiled program are grounded (lparse), and the test is run with the smodels inference engine.

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 program uses for reading the BCRL program and the time used for computation. *Non smodels time* includes the time to start a test case, to 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 4. The time needed to translate PCML to WCRL is shown ("PCML to WCRL (s)"). In addition, all configuration models were run once on smodels to find all configurations of each model without any requirements. The number of configurations ("#Configs") af-

Table 4

Run time characteristics of configuration models

Model	PCML to WCRL (s)	#Configs	Smodels (s)	Configs (1/s)	Non Smodels (s)
Compr FM	8.2	1,841,356,800	184,01.4	100,066	0.12
Compr FS	8.0	36,106,560	377.1	95,748	0.13
Compr FX	5.0	1,136,160	17.0	66,833	0.14
Vehicle	1.3	268,800,000	2,537.2	105,944	0.12

Table 5

1 Compr FM compressor results with test cases

#req	#sat	Find first (s)	#cfgs/case	#cfgs (1/s)	Unsat (s)
2	89	0.37	189,441,067	88,238	0.30
4	61	0.35	18,987,439	76,849	0.28
6	25	0.34	2,234,799	72,687	0.29
8	9	0.33	211,432	19,957	0.28
10	4	0.31	1,920	263	0.29
12	1	0.32	15,552	526	0.29
14–28	0	–	–	–	0.30

Table 6

Vehicle results with test cases

#req	#sat	Find first (s)	#cfgs/case	#cfgs (1/s)	Unsat (s)
2	95	0.05	37,317,928	98,238	0.04
4	85	0.05	5,855,831	88,913	0.04
6	59	0.05	747,205	84,642	0.05
8	33	0.05	108,638	71,394	0.05
10	22	0.09	29,136	54,358	0.07
12	8	0.07	9,526	42,361	0.08
14	4	0.06	289	3,853	0.08
16	0				0.08
18	2	0.06	9	140	0.07
20–24	0				0.06

ter symmetry breaking matches the manually analyzed expected number of configurations. Further, the smodels duration ("Smodels (s)") and the rate of configurations found per second ("#Configs (1/s)") are given. "Non smodels (s)" is averaged non smodels time from running the test cases.

Tables 5 and 6 show our main results from running the generated random test cases regarding the largest compressor model and the vehicle. 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 (s)" gives the average smodels duration of finding one configuration that satisfies the requirements, and "Unsat (s)" 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 (1/s)"). Non Smodels time from Table 4 can be added to get the average total duration of finding the first configuration or determining unsatisfiability.

The test arrangement caused occasional random delays of approximately $\frac{1}{2}$ s, possibly due to garbage collection in the Java environment, the functions of the operating system, or the virus scanner. Therefore, maximum durations are not shown. The maximum Smodels time for finding one configuration or determining unsatisfiability was still below 0.7 s. When repeated, the times were close to the average – typically, approx-

imately within 20% of the average, except for the vehicle model, where average duration was always less than 0.1 s, causing small absolute errors to show major relative differences.

4.3.5. Additional performance evaluations

The first author of this work has configured all the characterized products using the WeCoTin user interface (Linux only partially). Performing the configuration with the all configuration models except Linux is of acceptable speed. The web interface is slowest on the Broadband model before an attribute "area" with 436 possible values was fixed. A 2.4 GHz Intel Core 2 duo laptop responds in slightly over 3 s, reducing to less than a second after setting the largest domain attribute. All other configuration models can be configured with a feeling of instant response.

Compilation time from PCML to WCRL and then to BCRL is also acceptable. A script that compiles all the mentioned configuration models, except Linux, and a few additional test and sample models runs in 32 s with the above-mentioned laptop.

5. Discussion

5.1. Logic-based configurators in previous work

Previous work includes configurators applying numerous problem solving methods (see Section 1.3). This work showed that Weight Constraint Rules and the answer set programming paradigm are a feasible alternative to constraint satisfaction in general purpose configurators. Previously Weight Constraint Rules were applied to directly model individual configuration problems [49]. WeCoTin differs due to the general purpose, multi-domain approach, and high-level configuration specific modeling concepts with corresponding tools for modeling.

Numerous other logic-based methods have been applied in configurators. These include description logics (DLs), e.g., [2,25]. The role of description logics as an inference engine for supporting the actual configuration task is not completely clear. According to [25], the CLASSIC system was used to deduce logical implications (deductive closure) of the basis user inputs. According to [63], this has been augmented in case of larger products with external search algorithms and special purpose algorithms, and description logic was used to perform integrity checking of the results.

Constraint logic programming has been applied for configuration problems [39]. The implementation was based on the ECLiPSe constraint logic programming environment using the Finite Domains (FD) library. A high-level language for modeling configuration knowledge based on concepts similar to a subset of [44] was developed. Taxonomical hierarchy of component types, their attributes, ports and part relationships were included in the language. Configuration models expressed in this language were then translated into CLP programs, which enabled configuration based on expressed requirements. Results indicative of adequate performance in automatic configuration were achieved. This work bears many similarities to our approach. However, detailed comparison is difficult due to a lack of details. We go further with our empirical evaluation of the system and efficacy of modeling.

Most configurators infer consequences of configuration decisions and check consistency during the configuration process. WeCoTin checks "behind the scenes" during the configuration process that there exists a way to complete the configuration. We are not aware of other configurators that provide this functionality.

5.2. Configuration modeling and performance testing

Modeling efficacy results are subject to author bias, because modeling has only been performed by researchers involved in WeCoTin development.

When configuration models are characterized in previous work, usually the number of component types and/or connections is specified, e.g., [12,22,39,49]. We are not aware of previous work with deeper characterization of configuration models, such as application of inheritance, characterization of compositional structure, and other modeling mechanisms.

We are only aware of limited configurator performance testing in previous work; Syrjänen [49] 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 excluded or included 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 on a 233 MHz Intel Pentium II. The configuration duration was approximately the same as in our largest tested model (Compr FM) (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. However, our modeling was performed on a

higher-level conceptual model that does not offer opportunities for manual tweaking of performance.

Sharma and Colomb [39] developed a constraint logic programming (CLP) based language for configuration and diagnosis tasks. Experimental results stem from a thin Ethernet cabling configuration. The largest 12-node configuration included 126 port connections and required 12 s of CPU time on a dual 60 MHz SuperSparc processor-based system to find a configuration [39]. Direct performance comparison to our work is difficult due to port and connection oriented domain, different processor power, and missing details.

Mailharro [22] 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. The case product is larger and more complex than ours. Direct performance comparison is not possible due to limited details available.

Our work differs from previous work because we applied configuration models of several products for testing and generated varying numbers of random requirements that could reveal phase transition behavior. Furthermore, some previous work conducted performance evaluation with significantly larger products than our systematically tested products.

Systematic testing and ad-hoc results indicate adequate performance with the case products. There were no test cases with repeatable significantly inferior performance. In addition, there was no significant change of performance as a function of the number of requirements. The average configurations per second results weaken with increasing number of requirements. However, this is 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.

No critically constrained problems were found and no phase transition behavior was apparent. As expected, the number of configurations decreases exponentially as the number of requirements increases. Minor exceptions due to random requirements were encountered in the two models discussed above.

The CLib configuration benchmarks library contains 21 configuration problems [48]. These benchmarks include our systematically tested cases. We selected to test performance only on our own models whose semantics we know and that are expressed in a form that facilitates PCML modeling with the original problem structure. This is not the case for problems expressed in conjunctive normal form, such as those integrated to cLib based on [40].

5.3. Future work

Modeling more challenging configuration tasks such as telecommunications networks would enhance evaluation of WeCoTin. However, adequate modeling support for such cases would probably require extensions for WeCoTin to support connection and resource oriented configuration modeling concepts. Significant user interface changes would result, both in the Modeling Tool and especially in the Configuration Tool.

Systematic performance testing with a larger set of configuration models remains future work, although even this subset is larger than in most previous work. However, we do not expect phase transition behavior in already modeled cases – one reason why systematic tests were omitted for all but the first four finalized configuration models.

Modeling by product experts would significantly enhance the evaluation of practicality of the Modeling Tool. Further, it would be interesting to analyze how the application of more advanced mechanisms such as inheritance would change according to the background of the modelers.

Numerous potential system extensions would enhance the practicality of WeCoTin. These include maintenance interfaces for price lists (editing, importing, updating) and calculations, system setup wizards, extensions to the table constraint editor to support more complex expressions and ordered tables, implementation of resource and layout inheritance, advanced document generation facilities, and generic support for visualization of a configuration.

From the end-user point of view, numerous enhancements are possible. Optimization support, e.g., towards price, would be useful. Enhanced ways to express requirements such as acceptable ranges, minimums and maximums of attribute values, or specifying an abstract type for later specialization would also be helpful. In addition, active personalized recommendations and suggestions for fixing an inconsistent configuration could be provided. Further, management of defaults could be improved in several ways, including dynamic defaults and optimization to minimize the number or utility of violated soft constraints or the number of removed defaults to restore the consistency of a configuration. Explicit automatic completion of a configuration could sometimes be useful, although completion achieved via defaults is often appropriate. Further, ability to trigger informative messages based on the state of a configuration would be valuable. Finally, some configurators apply user inter-

faces where separate phases of configuration tasks are clearly named and the current phase is clearly identified. Such a mechanism could be an addition or alternative to the configuration tree.

The Configuration Tool interface should be more flexible. Primarily, it should be possible to include questions related to several feature individuals on a question page. Further, formulating a dynamic sequence of questions depending on previous answers could be useful. Here, hiding questions or alternatives dynamically would be helpful. If individual questions become irrelevant due to some selections, it would be beneficial to be able to dynamically exclude them from the evaluation of completeness of a configuration.

Adding support and calculation mechanisms for floating point numbers would extend the scope of configuration tasks that could be supported. However, we are not aware of methods that would retain sound and complete inference when such facilities are exploited.

6. Answers to research questions and conclusions

The main contribution of this work is the instantiation type artifact WeCoTin configurator. It incorporates a number of novel technical aspects, including sound and complete inference, high-level, object-oriented modeling based on a well-founded conceptual model, semi-automatic generation of user interfaces, and several aspects that ease long-term management. The construction and its evaluation provide positive answers to research questions Q1–Q3.

We conclude, based on modeling (and configuring) without difficulties and with an acceptable conceptual match using the sales view of real-world products (14 fully, 8 partially), that the function-oriented subset of the configuration conceptualization [44] is useful for modeling configuration knowledge (Q1). However, our current view is that given otherwise the same modeling mechanisms (compositional structure, taxonomy including inheritance, attributes, and constraints), it is largely irrelevant what the basic objects are called: 'features' as in WeCoTin, 'components' as in an earlier version of WeCoTin, 'functions' as in the conceptual model, or just 'objects'.

A configurator was constructed based on the idea of translation of configuration knowledge into Weight Constraint Rules [41,43]. It was applied to model and configure a number of real-world cases from several domains, including demonstrably adequate performance with the systematically tested models and ad-hoc manual configuration with other configuration models. Therefore, the approach can provide a practically feasible basis for a product configurator (Q2). Thus, industrially relevant configuration problems can be effectively modeled and configured with a configurator constructed based on the conceptual model and the Weight Constraint Rule based approach (Q3). However, an exception to sufficient performance is the very large Linux model, where achieving sufficient performance would require at least the capability to control when full inference is performed, and possibly other optimizations. We expect that the adequate performance of WeCoTin could be generalized to many other products suitable for web-based sales configuration. As a side-result, suitability of Smodels to work as an efficient inference engine for a general-purpose configurator was demonstrated.

Inference based on the stable models semantics of logic programs can provide a basis for constructing a practical and applicable configurator. The compilation of configuration models to Weight Constraint Rules is modular so that a small change to a configuration model causes only a small change to the Weight Constraint Rule representation. In addition, it follows a *general groundedness principle* of configuration knowledge representation: anything that is not allowed by the configuration model cannot hold in a correct configuration. In WCRL, there is no need to add so-called "completion" or "frame" axioms that forbid any other state of affairs from holding in a configuration other than those allowed by the configuration model, such as extra objects. This leads to a compact formalization of the configuration knowledge.

The characterizations provided in Section 4.1 form a proposal for basic static configuration model characterization (Q4a). Modeling mechanisms including compositional structure with cardinality and named parts (subfeature definitions), and explicit allowed types was useful. Attributes of feature types were the main modeling mechanism. Surprisingly, there was no use for large cardinalities in our modeled cases, although it is easy to imagine cases where large cardinalities can be applied, e.g., elevator doors or signaling equipment for different floors. There was no use for multiple inheritance, resource-based modeling or topological modeling, such as ports (Q4b). A method was developed that simulates a naïve user entering random requirements to a configurator equipped with a real-world configuration model (Q4c). The method could be applied to other types of configurator implementations.

Finally, WeCoTin has been commercialized, which constitutes market-based validation of pragmatic rel-

evance (“weak market test”) [20] of a construction. Kasanen et al. view that ideas and constructs compete in markets, and commercial adoption contributes to their significance. In their view “...even the weak market test is relatively strict – it is probably not often that a tentative construction is able to pass it”.

Acknowledgements

This work has been supported by Technology Development Centre of Finland. We thank Asko Martio, Ilkka Niemelä, Juha Nurmilaakso, Mikko Pasanen, Hannu Peltonen, Kati Sarinko and Reijo Sulonen for their valuable efforts and co-operation. Finally, we thank Gardner Denver Oy for sharing the configuration knowledge.

References

[1] A. Anderson, Towards tool-supported configuration of services, M.Sc. thesis, Department of Computer Science and Engineering, Helsinki University of Technology, Espoo, Finland, 2005.

[2] F. Baader, Description logics, in: *Reasoning Web. Semantic Technologies for Information Systems*, S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M. Rousset and R.A. Schmidt, eds, Springer, 2009, pp. 1–39.

[3] B. Bos, H.W. Lie, C. Lilley and I. Jacobs, Cascading style sheets, level 2, CSS2 Specification, 1998, retrieved 2010-03-30, available at: http://www.w3.org/TR/2008/REC-CSS2-20080411.

[4] R. Cunis, A. Günter, I. Syska, H. Peters and H. Bode, PLAKON – An approach to domain-independent construction, in: *Proceedings of the Second International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-89)*, 1989, pp. 866–874.

[5] cyLEDGE, International configurator database, cyLEDGE Media GmbH, Retrieved 2009-11-27, available at: http://www.configurator-database.com/services/configurator-database.

[6] S.R. Damiani, T. Brand, M. Sawtelle and H. Shanzer, *Oracle Configurator Developer User's Guide, Release 11i*, Oracle Corporation, 2001.

[7] R.P. Desisto, Constraints still key for product configurator deployments, Gartner Research Report, T-22-9419, Gartner, Inc., 2004.

[8] ECMA, ECMAScript language specification – Standard ECMA-262, 3rd edn, 1999.

[9] A. Felfernig, G. Friedrich, D. Jannach and M. Zanker, Intelligent support for interactive configuration of mass-customized products, in: *Proceedings of 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2001*, 2001, pp. 746–756.

[10] A. Felfernig, G.E. 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** (2000), 449–469.

[11] A. Felfernig, M. Schubert and M. Mandl, Personalized diagnoses for inconsistent user requirements, *AI EDAM* **25** (2011), 175–183.

[12] G. Fleischanderl, G.E. Friedrich, A. Haselböck, H. Schreiner and M. Stumptner, Configuring large systems using generative constraint satisfaction, *Intelligent Systems and Their Applications, IEEE* **13** (1998), 59–68. (See also *IEEE Intelligent Systems*.)

[13] F. Frayman and S. Mittal, COSSACK: A constraint-based expert system for configuration tasks, in: *Knowledge-Based Expert Systems in Engineering: Planning and Design*, 1987, pp. 143–166.

[14] A. Haag, Sales configuration in business processes, *IEEE Intelligent Systems and Their Applications* **13** (1998), 78–85.

[15] A. Haag, “Dealing” with configurable products in the SAP Business Suite, in: *Papers from the Configuration Workshop at IJCAI'05*, D. Jannach and A. Felfernig, eds, 2005, pp. 68–71.

[16] A. Haag, U. Junker and B. O'Sullivan, Explanation in product configuration, *IEEE Intelligent Systems* **22** (2007), 83–85.

[17] M. Heinrich and E.W. Jüngst, A resource-based paradigm for the configuring of technical systems from modular components, in: *Proceedings of Seventh IEEE Conference on Artificial Intelligence Applications*, 1991, pp. 257–264.

[18] M. Heiskala, K. Paloheimo and J. Tiihonen, Mass customization with configurable products and configurators: A review of benefits and challenges, in: *Mass Customization Information Systems in Business*, 1st edn, T. Blecker and G. Friedrich, eds, IGI Global, 2007, pp. 1–32.

[19] A.R. Hevner, S.T. March, J. Park and S. Ram, Design science in information systems research, *MIS Quarterly* **28** (2004), 75–105.

[20] E. Kasanen, K. Lukka and A. Siitonen, The constructive approach in management accounting research, *Journal of Management Accounting Research* **5** (1993), 243–264.

[21] A.K. Mackworth, Consistency in networks of relations, *Artificial Intelligence* **8** (1977), 99–118.

[22] D. Mailharro, A classification and constraint-based framework for configuration, *AI EDAM* **12** (1998), 383–397.

[23] J. McDermott, R1: A rule-based configurator of computer systems, *Artificial Intelligence* **19** (1982), 39–88.

[24] J. McDermott, R1 (“XCON”) at age 12: Lessons from an elementary school achiever, *Artificial Intelligence* **59** (1993), 241–249.

[25] D.L. McGuinness and J.R. Wright, An industrial-strength description logic-based configurator platform, *IEEE Intelligent Systems and Their Applications* **13** (1998), 69–77.

[26] S. Mittal and B. Falkenhainer, Dynamic constraint satisfaction problems, in: *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, 1990, pp. 25–32.

[27] S. Mittal and F. Frayman, Towards a generic model of configuration tasks, in: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI)*, 1989, pp. 1395–1401.

[28] O. Najmann and B. Stein, A theoretical framework for configuration, in: *Proceedings of the IEA/AIE'92, 5th Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Belli, ed., 1992, pp. 441–350.

[29] J. Nielsen, Usability engineering, in: *Anonymous*, 1993.

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

[31] J. Nurmilaakso, WeCoTin.calc documentation, Unpublished project documentation, Espoo, Finland, 2004.

[32] M. Pasanen, Warnings and pre-selection packages in a weight constraint rule based configurator, M.Sc. (Eng.) thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2003.

[33] H. Peltonen, J. Tiihonen and A. Anderson, Configurator tool concepts and model definition language, Unpublished working document of Helsinki University of Technology, Software Business and Engineering Institute, Product Data Management Group, Espoo, Finland, 2001.

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

[35] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1999.

[36] D. Sabin and R. Weigel, Product configuration frameworks – A survey, *IEEE-Intelligent-Systems* **13** (1998), 42–49.

[37] F. Salvador, P.M. de Holan and F.T. Piller, Cracking the code of mass customization, *MIT Sloan Management Review* **50** (2009), 71–78.

[38] D.B. Searls and L.M. Norton, Logic-based configuration with a semantic network* 1, *The Journal of Logic Programming* **8**(1-2) (1990), 53–73.

[39] N. Sharma and R. Colomb, Mechanising shared configuration and diagnosis theories through constraint logic programming, *The Journal of Logic Programming* **37** (1998), 255–283.

[40] C. Sinz, A. Kaiser and W. Kuchlin, Formal methods for the validation of automotive product configuration data, *AI EDAM* **17** (2003), 75–97.

[41] T. Soininen, An approach to knowledge representation and reasoning for product configuration tasks, Ph.D. thesis, Helsinki University of Technology, Department of Computer Science and Engineering, 2000.

[42] T. Soininen, E. Gelle and I. Niemelä, A fixpoint definition of dynamic constraint satisfaction, in: *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, J. Jaffar, ed., Springer-Verlag, London, UK, 1999, pp. 419–433.

[43] T. Soininen, I. Niemelä, J. Tiihonen and R. Sulonen, Representing configuration knowledge with weight constraint rules, in: *Proceedings of the AAAI Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge*, 2001, pp. 195–201.

[44] T. Soininen, J. Tiihonen, T. Mannisto and R. Sulonen, Towards a general ontology of configuration, *AI EDAM* **12** (1998), 357–372.

[45] M. Stumptner, An overview of knowledge-based configuration, *AI Communications* **10** (1997), 111–125.

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

[47] M. Stumptner, A. Haselbock and G. Friedrich, COCOS – A tool for constraint-based, dynamic configuration, in: *Proceedings of the Tenth Conference on Artificial Intelligence for Applications*, 1994, pp. 373–380.

[48] S. Subbarayan, CLib: Configuration benchmarks library, available at: http://www.itu.dk/research/cla/externals/clib.

[49] T. Syrjänen, Including diagnostic information in configuration models, in: *Proceedings of the First International Conference on Computational Logic*, 2000, pp. 837–851.

[50] T. Syrjänen, Software/smodels/lparse.ps.gz, Lparse 1.0 User's Manual, 1.0, Espoo, Finland, 2002.

[51] J. Tiihonen, Computer-assisted elevator configuration, M.Sc. (Eng.) thesis, Department of Computer Science, Helsinki University of Technology, Espoo, Finland, 1994.

[52] J. Tiihonen, National product configuration survey – Customer specific adaptation in the Finnish industry, Licentiate of Technology Thesis, Helsinki University of Technology, Department of Computer Science, Laboratory of Information Processing Science, Espoo, 1999.

[53] J. Tiihonen, Characterization of 26 configuration models, in: *Proceedings of the IJCAI-09 Workshop on Configuration (ConfWS-09)*, M. Stumptner and P. Albert, eds, 2009, pp. 69–76.

[54] J. Tiihonen, Characterization of 26 configuration models, in: *Proceedings of the IJCAI-09 Workshop on Configuration (ConfWS-09)*, M. Stumptner and P. Albert, eds, pp. 69–76.

[55] J. Tiihonen, M. Heiskala, K. Paloheimo and A. Anderson, Applying the configuration paradigm to mass-customize contract based services, in: *Extreme Customization: Proceedings of the MCPC 2007 World Conference on Mass Customization & Personalization*, W.J. Mitchell, F.T. Piller, M. Tseng, R. Chin and B.L. McClanahan, eds, 2007, Paper ID MCPC-134-2007, Section 7.5.3.

[56] J. Tiihonen and T. Soininen, State-of-the-practice in product configuration – A survey of 10 cases in the Finnish industry, in: *Knowledge Intensive CAD*, T. Tomiyama, M. Mäntylä and S. Finger, eds, London, 1996, pp. 95–114.

[57] J. Tiihonen and T. Soininen, Product configurators – Information system support for configurable products, Tech. Rep. TKO-B137, Helsinki University of Technology, Espoo, 1997.

[58] J. Tiihonen, T. Soininen, T. Mannisto and R. Sulonen, Configurable products – Lessons learned from the Finnish industry, in: *Proceedings of the 2nd International Conference on Engineering Design and Automation (ED&A'98)*, 1998 (CD).

[59] J. Tiihonen, T. Soininen, I. Niemelä and R. Sulonen, Empirical testing of a weight constraint rule based configurator, in: *Proceedings of the Configuration Workshop, 15th European Conference on Artificial Intelligence*, 2002, pp. 17–22.

[60] J. Tiihonen, T. Soininen, I. Niemelä and R. Sulonen, A practical tool for mass-customising configurable products, in: *Proceedings of the 14th International Conference on Engineering Design*, 2003, paper number 1290 (CD).

[61] J. Tikkala, S. Ailus, P. Hakkarainen, M. Koskimäki, J. Parviainen and P. Ranta, Final Report, T-76.115: Dotcomrades (In Finnish: Loppuraportti, T-76.115: Dotcomrades), Espoo, Finland, Retrieved 2009-10-12, available at: http://www.soberit.hut.fi/T-76.115/02-03/palautukset/groups/Dotcomrades/de/loppuraportti.html.

[62] B. Wielinga and G. Schreiber, Configuration-design problem solving, *Expert, IEEE [See also IEEE Intelligent Systems and Their Applications]* **12** (1997), 49–56.

[63] J.R. Wright, D.L. McGuinness, C.H. Foster and G.T. Vesonder, Conceptual modeling using knowledge representation: Configurator applications, in: *Proceedings of the Workshop on Artificial Intelligence in Distributed Information Networks, IJCAI-95*, 1995.