# Integration of Distributed Constraint-Based Configurators

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

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

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

## 1 Introduction

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

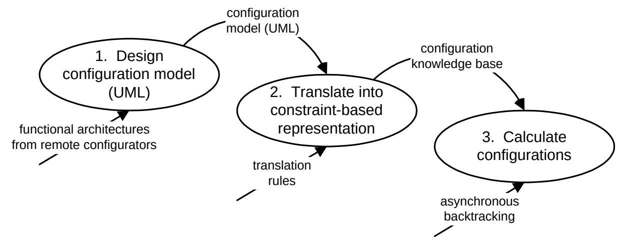
Especially the integration of configurators in order to support cooperative configuration such as supply chain integration of customizable products is an open research issue. Current configurator approaches [7] are designed for solving local configuration problems, but there is still no support for cooperative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

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

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

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

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



1. Design configuration model (UML)
configuration model (UML)
2. Translate into constraint-based representation
configuration knowledge base
3. Calculate configurations
functional architectures from remote configurators
translation rules
asynchronous backtracking

**Figure 1.** Configuration system development process

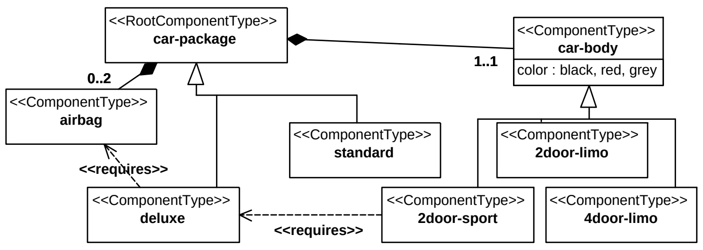
## 2 Designing configuration knowledge bases

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

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

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

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



<<RootComponentType>> car-package
<<ComponentType>> car-body
color : black, red, grey
1..1
0..2
<<ComponentType>> airbag
<<ComponentType>> standard
<<ComponentType>> 2door-limo
<<requires>>
<<ComponentType>> deluxe
<<ComponentType>> 2door-sport
<<ComponentType>> 4door-limo
<<requires>>

**Figure 2.** Component structure of car configurator

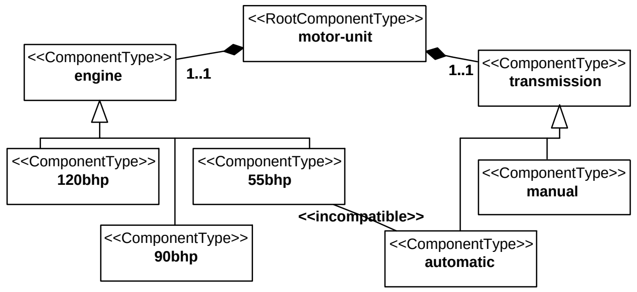
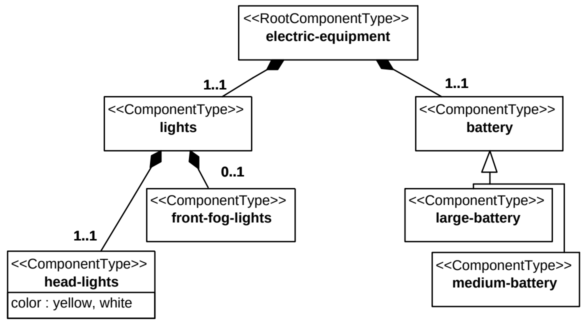


**Figure 3.** Component structure of motor configurator



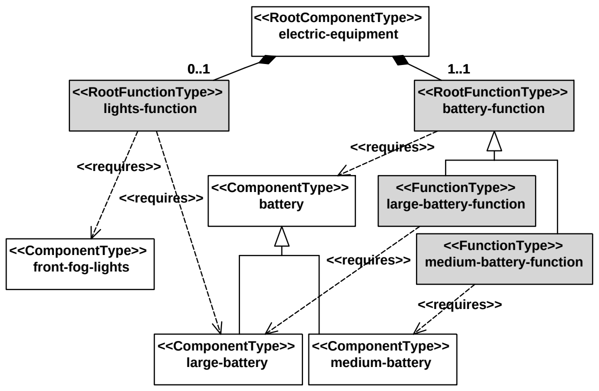
<<RootComponentType>> electric-equipment
1..1
1..1
<<ComponentType>> lights
<<ComponentType>> battery
0..1
<<ComponentType>> front-fog-lights
<<ComponentType>> large-battery
1..1
<<ComponentType>> head-lights
color : yellow, white
<<ComponentType>> medium-battery

**Figure 4.** Component structure of electric equipment configurator

## 3 Distributed Configuration Task

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

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



<<RootComponentType>> electric-equipment
0..1
1..1
<<RootFunctionType>> lights-function
<<RootFunctionType>> battery-function
<<requires>>
<<requires>>
<<requires>>
<<ComponentType>> battery
<<FunctionType>> large-battery-function
<<ComponentType>> front-fog-lights
<<requires>>
<<FunctionType>> medium-battery-function
<<requires>>
<<ComponentType>> large-battery
<<ComponentType>> medium-battery

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4 Variable Partitioning for Asynchronous Backtracking

### 4.1 Distributing Functional Architectures

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

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

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

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

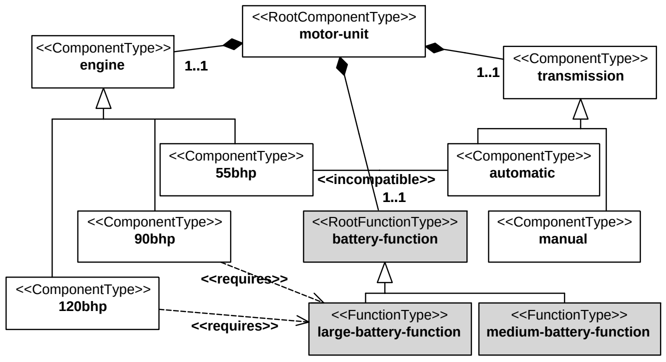


**Figure 6.** Imported functional architecture of motor-unit configurator

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

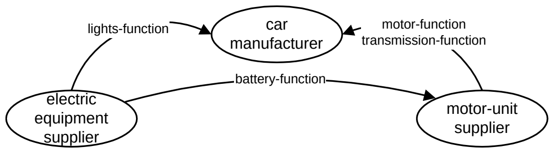


**Figure 7.** Distribution of functional architectures

### 4.2 Asynchronous Backtracking for Distributed Configuration

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

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

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

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

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

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

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

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

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

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

## 4.3 Example for Distributed Configuration

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

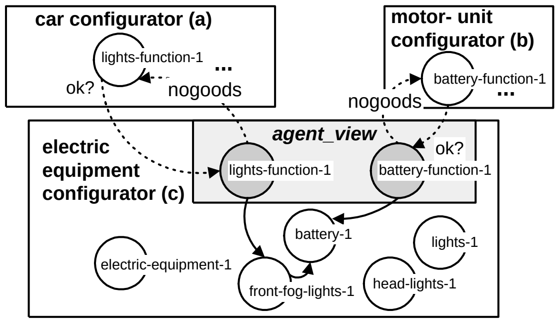


**Figure 8.** Interface of electric equipment configurator

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

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

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

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

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

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

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

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

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

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

## 5 Related Work

There is a broad spectrum of representation formalisms employed in knowledge-based configuration systems [1], [10], [12], [18], [19]. Todays configurators are tailored to solve local configuration tasks and there is no support for integrating these systems in order to allow cooperative configuration. Furthermore, the increasingly complex tasks tackled by configuration systems require the provision of methods for designing configuration knowledge bases on a higher level of abstraction also understandable by technical experts.

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

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

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

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

## 6 Conclusions

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

## REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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