Integrating Knowledge-Based Configuration Systems by Sharing Functional Architectures

Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker

Institut für Wirtschaftsinformatik und Anwendungssysteme, Produktionsinformatik,
Universitätsstrasse 65-67, A-9020 Klagenfurt, Austria,
email: felfernig@ifi.uni-klu.ac.at
tel. ++43/463/2700/6204

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.). A configuration task can be characterized through a set of components, a description of their properties (attributes and connection points), and constraints on legal configurations. Given some customer requirements, the result of computing a configuration is a set of components, corresponding attribute valuations, and connections satisfying all constraints and customer requirements.

Here we will employ a scenario where several configuration systems jointly solve such a configuration task. The need for distributed configuration problem solving arises out of an economic necessity. Supply chain integration for complex customizable products and services demands an integration of local configuration systems along the value chain of these products. A centralized approach with a single configurator, that comprises the configuration knowledge of the final product and is additionally capable to configure the supplied parts, is only a theoretical alternative to a cooperative approach. The reasons lie in organizational and corporate security concerns of the involved business entities. As the domain experts are distributed over different manufacturers the maintenance of the configuration knowledge has to happen decentralized within each supplier. Further no supplying organization wants to share her whole configuration knowledge including technical details with the buying manufacturer for reasons of privacy and competition. However current configuration technology [7] encompasses only the solving of local configuration problems, and distributed configuration is still an open research issue.

In this paper we will show how our framework that supports the conceptual design of configuration knowledge bases and the automatic translation of the resulting models into an executable logic representation described in [8] and [9] can be extended in order to support distributed configuration problem solving within our prototype environment. In [8] we present how to employ UML - Unified Modeling Language [23], which is a standard design language widely applied in industrial software development processes, to graphically develop configuration knowledge bases following the component port paradigm described by [16]. We were using the built-in extension mechanism of UML to define configuration domain specific stereotypes based on the work of [25] on a general ontology[1]for configuration modeling. Further we show there how to automatically translate the graphically represented configuration knowledge into a machine executable formal representation. In [9] we give additional translation rules for those complex constraints that cannot be represented with the graphical concepts of UML and have to be formulated with OCL[2] (Object Constraint Language). The acquisition of functional configuration knowledge with our framework is described in [10]. Functional configuration knowledge [3], [4], [16] determines *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* the functions are implemented).

Here our approach towards a distributed configuration task is based on the integration of the different configuration models of cooperating business entities by a mechanism of shared functional architectures. This proposed development process for cooperatively solvable configuration tasks is outlined in Figure 1. In the first phase the locally applicable configuration knowledge is modeled and the functional architectures from cooperating configuration systems are integrated, 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

[1] We interpret ontologies in the sense of [5], i.e. ontologies are theories about the sorts of objects, properties of objects, and relations between objects that are possible in a specified domain of knowledge.

[2] The object oriented expression language OCL is part of the UML standard.

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). In the following the configuration system is employed in productive use, i.e. solutions for a distributed configuration task are calculated (phase 3).

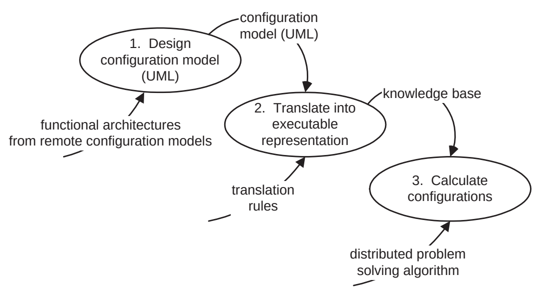
1. Design configuration model (UML)
configuration model (UML)
functional architectures from remote configuration models
2. Translate into executable representation
knowledge base
translation rules
3. Calculate configurations
distributed problem solving algorithm

Fig. 1. Configuration system development process

The rest of the paper is organized as follows. First, we sketch the conceptual design of an example 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 [16] and show how to translate the example model into this formalism. In Section 4 we show how to share functional architectures between configuration systems and how to organize the local configuration knowledge to be exploited by a distributed problem solving algorithm. 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). In Section 5 we discuss our approach and describe the prototypical implementation of our approach towards cooperative configuration systems using commercial tools. Finally we cite related work followed by general conclusions.

2 A General Configuration Ontology

For presentation purposes we introduce simplified UML models of a car manufacturer (Figure 2), a motor supplier (Figure 3), and an electric equipment supplier (Figure 4) as a working example. These diagrams represent the generic product structure, i.e. all possible variants of the product. The set of possible products is restricted through a set of constraints which relate to customer requirements, technical restrictions, economic factors, and restrictions according to the production process.

A simple scenario for solving a configuration task could be the following. The customer contacts the car manufacturer and communicates requirements concerning the car configuration *(car-body:4door-limo, car-package:standard, engine:55bhp, transmission:manual, front-fog-lights)*. The car configurator calculates a local solution and contacts the motor-unit configurator for configuring a *55-bhp/manual motor-unit*. Furthermore, the car configurator contacts the electric equipment configurator for providing an *electric-equipment* containing *front-fog-lights* which represent optional parts in the configuration model of the electric equipment supplier (see Figure 4). The motor unit configurator contacts the electric equipment configurator for configuring a battery. Finally, the calculation of the distributed configuration is finished and the result is presented to the customer. Note that the customer requirements are related to the configuration model of the car manufacturer as well as to the models of the electric equipment supplier and the motor-unit supplier, i.e. the car manufacturer needs further information from the suppliers in order to provide the relevant information for the customer. In the following we will show how this information is provided by exchanging functional architectures.

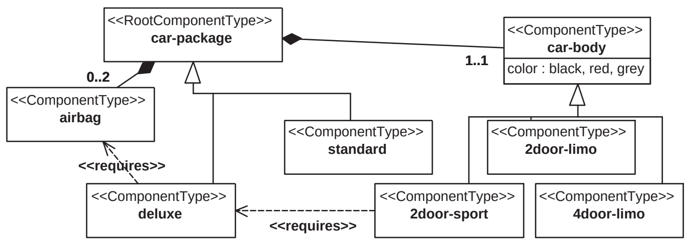

Fig. 2. Component structure of car configurator

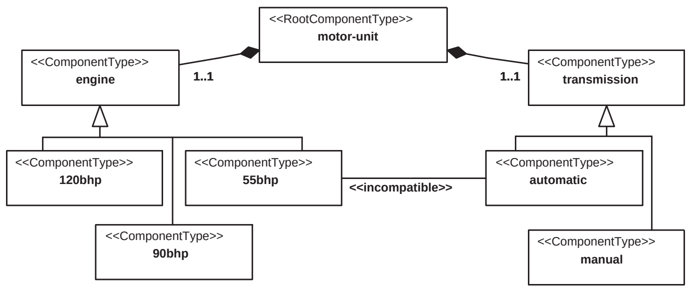

Fig. 3. Component structure of motor configurator

In order to make configuration models executable, we propose a translation into the component port representation, which is well established for modeling and solving configuration problems [16]. In general, consistency-based tools build onto this model can use the logic theory derived from the UML configuration model.

We employ the extension mechanism of UML (stereotypes) to express domain-specific modeling concepts, which has been shown to be a promising approach in other areas [22]. The semantics of the different modeling concepts are formally defined by the mapping of the graphical notation to logical sentences based on

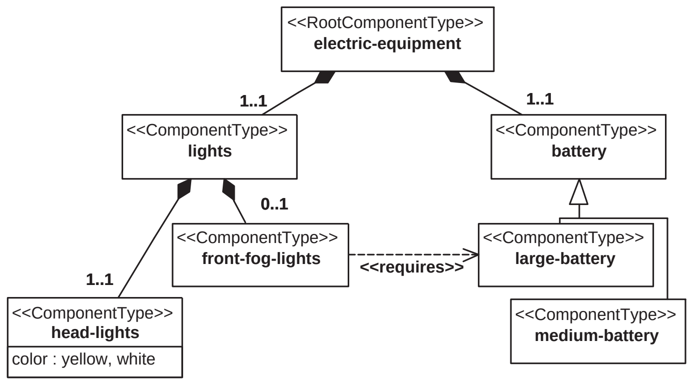

Fig. 4. Component structure of electric equipment configurator

the component port model (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 the basic parts of the ontology employed for designing configuration models [25].

- **Component types** These represent parts the final product can be built of. Component types are characterized by attributes (e.g. *car-body* and *color* in Figure 2).
- **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 (e.g. *lights-function* in Figure 5).
- **Resources** Parts of a configuration problem can be seen as a resource balancing task, where some of the component (function) types *produce* some resource and others are *consumers.* E.g. the maximum *price* of the car must not exceed a certain limit - in this case the different components (functions) of a configuration represent consumers, the component storing the maximum *price* represents the producer.
- **Generalization** Component (function) types with a similar structure are arranged in a generalization hierarchy (e.g. *engine* in Figure 3).
- **Aggregation** Aggregations between components (functions) represented by part-of structures state a range of how many subparts an aggregate can consist of (e.g. a *transmission* is part of a *motor-unit* - see Figure 3).
- **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 with each other (e.g. a *radio power-supply* must be connected to a *battery*).
- **Compatibility and requirements relations** Some types of components (functions) cannot be used in the same final configuration - they are *incompatible* (e.g. *55bhp engine* is *incompatible* with *automatic transmission* - see Figure 3). In other cases, the existence of one component (function) *requires*

the existence of another special type in the configuration (e.g. *front-fog-lights requires large battery* - see Figure 4).

- **Functional architectures** Functional architectures represent exactly those parts of the configuration model, which can be shared between cooperating configurators (e.g. functional architecture *battery-function* in Figure 5). 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) expressions.
- **Additional modeling concepts and constraints** Constraints on the product model, which can not be expressed graphically, are formulated using the language OCL. 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 [21]. 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.

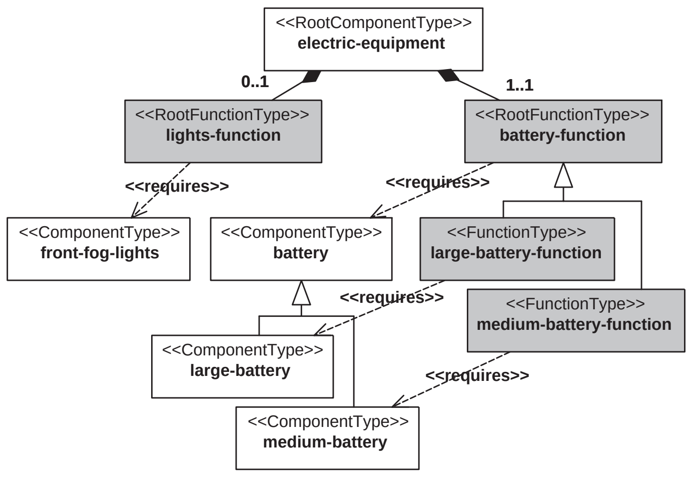

Fig. 5. Functional architectures and component mapping of electric equipment supplier

3 Distributed Configuration Task

In this section we give a formal definition of a distributed configuration task based on the component port model [16], which allows an intuitive definition using configuration domain specific representation concepts.

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 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 [12], [16]. 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 (all represented in UML). Based on this characterization of a local configuration task [12], [16] 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, 5) is the following[3] :

```
 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,
```

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

```
                    lights-function-port,battery-function-port}[4].
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 *front-fog-lights requires large-battery* (Figure 4) is translated as follows[5]:

```
type(ID1, front-fog-lights) ∧
    conn (ID1, lights-port, ID2, front-fog-lights-port) ∧
    conn (ID2, electric-equipment-port, ID3, lights-port) ⇒
      ∃(ID4) conn (ID4, electric-equipment-port, ID3, 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(head-lights-1, head-lights).
type(lights-1, lights).
type(battery-1, medium-battery).
func(battery-function-1, medium-battery-function).
conn(head-lights-1, lights-port, light-1, head-lights-port).
conn(lights-1, electric-equipment-port, electric-equipment-1, lights-port).
conn(battery-1, electric-equipment-port, electric-equipment-1, battery-port).
conn(battery-function-1, electric-equipment-port,
       electric-equipment-1, battery-function-port).
```

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

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

We specify that *FUNCS* includes all required functions, *COMPS* includes all required components, *CONNS* describes all required connections, and *ATTRS* includes a complete value assignment to all variables in order to achieve a complete distributed configuration[6]. Let AX_{comp} be the additional sentences for completeness purpose.

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

[5] 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.

[6] This is accomplished by additional logical sentences which can be generated using the domain description (see [12] for more details).

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. 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*.

4 Solving Distributed Configuration Tasks

4.1 Distribution of Functional Architectures

In order to enable effective distributed configuration, the configuration knowledge must be shared between configurators. In the following we will 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, 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 the 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 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.

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[7]. Figure 7 gives an overview of the distribution of functional architectures between the car manufacturer, electric equipment supplier, and the motor-unit supplier.

[7] Note that the functional architecture of the motor unit supplier exported to the car manufacturer as well as the functional architecture of the car manufacturer "exported" to the customer are not shown in our example models.

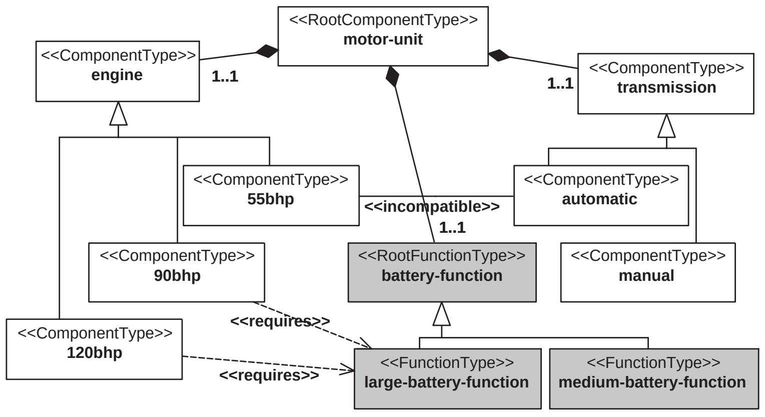
<<RootComponentType>> motor-unit
<<ComponentType>> engine
<<ComponentType>> transmission
1..1
1..1
<<ComponentType>> 55bhp
<<ComponentType>> automatic
<<incompatible>>
1..1
<<ComponentType>> 90bhp
<<RootFunctionType>> battery-function
<<ComponentType>> manual
<<requires>>
<<ComponentType>> 120bhp
<<requires>>
<<FunctionType>> large-battery-function
<<FunctionType>> medium-battery-function

Fig. 6. Imported functional architecture of motor-unit configurator

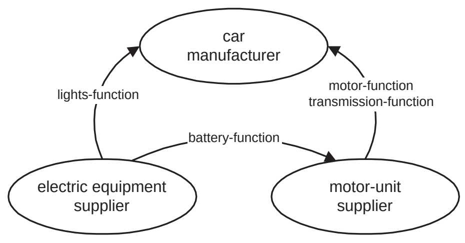

Fig. 7. Distribution of functional architectures

After having distributed the configuration knowledge by exchanging and integrating functional architectures we must define rules for how to organize the local configuration knowledge in order to employ a distributed problem solving algorithm for calculating a solution for a given distributed configuration task.

In the following we sketch the application of *asynchronous backtracking* proposed by [28], which is an algorithm calculating solutions for distributed constraint satisfaction problems (DCSP). In asynchronous backtracking problem variables are distributed among problem solving agents. Constraints concerning shared variables are directed between the agents in the sense that one of the connected agents is the value sending agent (agent, which instantiates the shared variables), the other one is the constraint evaluating agent which informs the value sending agent about inconsistencies of shared variables. Changes of shared variable assignments are communicated to corresponding constraint evaluating agents via *ok?* messages, inconsistent assignments are comunicated to value sending agents via *nogood* messages. Assignments of value sending agents are stored in the local *agent_view* of the constraint evaluating agent, which is used to check the consistency of instantiations of local variables with instantiations of value sending agents. *Nogoods* represent conflicting variable instantiations which are calculated by applying resolution. Value sending agents have a higher priority than connected constraint evaluating agents.

Asynchronous backtracking offers the basis for bounded learning strategies supporting the efficient revision of requirements and design decisions which is of particular interest for configuration systems, since supplier configurators eventually discover conflicting requirements *(nogoods)* which must be communicated back to the requesting configurator. Furthermore, this algorithm can easily be integrated into different configuration systems.

In order to be executeable by asynchronous backtracking the domain description DD_i of each configurator i must be organized conforming the following rules.

1. **Preventing infinite processing loops**[8] **:** If a functional architecture FA_{ij} of configuration model i is exported to configuration model k, no functional architecture from k can be integrated in i. When regarding the resulting configurators, *configurator i* is the supplier configurator and *configurator k* is the consumer configurator, i.e. *configurator i* is the constraint evaluating configurator and *configurator k* is the value sending configurator. Figure 8 shows the communication structure between the three example configurators.
2. **Constraint evaluating configurators:** Let F_j be the set of functions derived from function types of a functional architecture FA_{ij} of configuration model i imported from configuration model k *($k \neq i$)*, A_j the set of attributes derived from attributes attached to function types of FA_{ij}, and P_j the set of ports derived from aggregation relations between functions in FA_{ij}. Then each $V \in (F_j \cup A_j \cup P_j)$ has a *constraint evaluating configurator k,* i.e. is represented in the *agent_view* of configurator *k.*
3. **Value sending configurators:** Let F_j be the set of functions derived from function types of a functional architecture FA_{kj} of configuration model k exported to configuration model i *($i \neq k$)*, A_j the set of attributes derived from attributes attached to function types of FA_{kj}, and P_j the set of ports derived from aggregation relations between functions in FA_{kj}. Then each $V \in (F_j \cup A_j \cup P_j)$ is represented in the local *agent_view* of *configurator k* through a copy of *V*, which is updated by the *value sending configurator i.*

4.2 Example: Distributed Car Configuration

In order to illustrate the concepts discussed so far, we now give an example for solving a distributed configuration task using asynchronous backtracking. The car configurator must configure a car conforming the following functional requirements stemming from customer requirements[9]:

```
ok?((car-body-function-1,4door-limo-function),(car-package-1,standard-function),
    (engine-1,55bhp-function),(transmision-1, manual-function),
    (lights-function-1,lights-function)).
```

[8] Infinite processing loops are prevented in order to guarantee the termination of the algorithm.

[9] In order to keep the example simple, we omit the *conn* predicates describing the connections between the different functions.

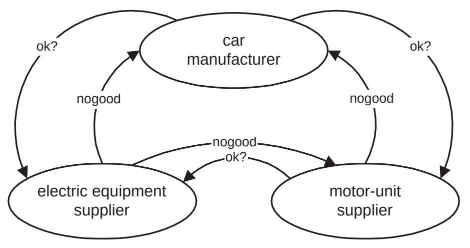

Fig. 8. Communication structure of example configurators

Regarding these functional requirements the car configurator calculates a local solution and propagates the functional requirements to the concerned outgoing configurators. The motor-unit configurator receives the following requirements:

ok?((engine-1,55bhp-function),(transmission-1,manual-function)).

The electric equipment configurator receives the following functional requirements from the car configurator:

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

The motor-unit configurator calculates a local solution regarding the given functional requirements (chooses a *medium-battery*) and communicates the following functional requirements to the electric equipment configurator:

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

The electric equipment configurator tries to calculate a local solution and detects a contradiction between *front-fog-lights-function* and *medium-battery-function*, since the *front-fog-lights* component implementing the corresponding function *requires* a *large-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 a *large-battery* component with the corresponding *large-battery-function*. The new functional requirements are communicated to the electric equipment configurator. Finally the electric equipment configurator calculates a solution regarding the requirements of the car configurator and the motor-unit configurator.

5 Discussion and Prototype Environment

Our work towards distributed configuration presented in this paper builds on our previous work on knowledge acquisition for configuration systems [8]. This approach differs from previous work on knowledge-engineering methodologies for configuration systems by avoiding the use of proprietary representation concepts. The Unified Modeling Language is a popular standardized conceptual modeling language that uses a graphical notation. This makes the problem domain more comprehensible and eases the communication between the people involved (e.g. domain experts and knowledge engineers). As UML allows to extend the language by defining additional modeling concepts (*stereotypes*), an automated translation from the conceptual model to an executable representation (e.g. logic sentences or constraint representation) is made possible. Thus the configuration knowledge can be maintained on the conceptual level.

When it comes to the design of a distributed configuration problem the task of knowledge sharing among different configuration systems can also be accomplished at an abstract level within our framework. Although for presentation purposes we employed the same constraint satisfaction problem representation for every example configurator, different knowledge representation formalisms resulting from different translation rules for each configuration system are possible. The choice of an algorithm for distributed problem solving depends therefor on the degree of cooperation between the configuration systems. In our example a very close interaction with a distributed backtracking algorithm could be shown, because all participating configurators employed a syntactically and semantically equivalent knowledge representation formalism.

The concepts presented in this paper are implemented in a prototype environment for the construction of cooperative configuration systems. For the design of product configuration models we employ the CASE tool Rational Rose. Our translation tool uses a XMI (XML Metadata Interchange [20]) representation (generated from Rational Rose models) as input and generates e.g. C++ code which includes the ILOG configuration libraries . We have evaluated our approach on real world problems from the domains of private telephone switching systems and automotive industry and noticed a significant reduction of development efforts. An extended version of the distributed car configuration problem presented in this paper is implemented using ILOG configurators which are implemented as CORBA objects. The communication between the configurators is realized using simple KQML [18] performatives. The message content is represented as an XML [27] document containing a set of functional requirements or a set of incompatible requirements.

6 Related Work

There is a long history in developing configuration tools in knowledge-based systems [26]. Progressing from rule-based systems like R1/XCON [1] higher level representation formalisms were developed, i.e. various forms of constraint satisfaction [11], description logics [15], or functional reasoning [24]. [14] propose a resource-based paradigm of configuration where the number of components of a particular type occuring in the configuration depends on the amount of that resource required.

An extensive framework for modeling configuration knowledge and the problem solving behaviour (VITAL methodology) can be found e.g. in [17]. The VITAL approach structures the process of development of a configuration model and an independent description of the problem solving strategy. Therefor this work is complementary to ours, where we start from a conceptual configuration model and translate to different machine executable representations.

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 [13]. In [2] 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 devided 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. [16] propose the intergration 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 [19] a KIF [18] 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 [6] 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.

7 Conclusions

The integration of businesses by internet technologies boosts the demand for distributed problem solving. In particular in knowledge-based configuration we have to move from stand-alone configurators to distributed configuration. In this paper we have proposed a framework for modeling configuration knowledge bases using a standard design language. The representation of (configuration) knowledge on a conceptual level is well suited as basis for the communication with technical experts. Furthermore, the automatic translation of the resulting configuration models significantly reduces the configuration system building effort. Based on these basic concepts we have proposed a framework for the integration of configuration systems based on the sharing of functional architectures which are an integrative part of a configuration model. 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.

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.H. Bourdeau and B.H.C. Cheng. A formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21,10:799–821, 1995.
3. B. Chandrasekaran. Functional Representation and Causal Processes. *Advances in Computers*, 38:73–143, 1994.
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. B. Chandrasekaran, J. Josephson, and R. Benjamins. What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, 14,1:20–26, 1999.
6. T.P. Darr and W.P. Birmingham. An Attribute-Space Representation and Algorithm for Concurrent Engineering. *AIEDAM*, 10,1:21–35, 1996.
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. A. Felfernig, G. Friedrich, and D. Jannach. Generating product configuration knowledge bases from precise domain extended UML models. In *12th International Conference on Software Engineering and Knowledge Engineering*, Chicago, USA, 2000.
10. A. Felfernig, D. Jannach, and M. Zanker. Diagrammatic Acquisition of Functional Knowledge for Product Configuration Systems with the Unified Modeling Language. In *International Conference on the Theory and Application of Diagrams*, Edinburgh, UK, 2000.
11. G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. In E. Freuder B. Faltings, editor, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13,4, pages 59–68. 1998.
12. G. Friedrich and M. Stumptner. Consistency-Based Configuration. In *AAAI Workshop on Configuration, Technical Report WS-99-05*, pages 35–40, Orlando, Florida, 1999.
13. 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.

14. E.W. J ngst M. Heinrich. A resource-based paradigm for the configuring of technical systems from modular components. In *Proc. 7th IEEE Conference on AI applications (CAIA)*, pages 257–264, Miami, FL, USA, 1991.
15. 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.
16. S. Mittal and F. Frayman. Towards a Generic Model of Configuration Tasks. In *Proc. of the 11th IJCAI*, pages 1395–1401, Detroit, MI, 1989.
17. E. Motta, A. Stutt, Z. Zdrahal, K. O Hara, and N. Shadbolt. Solving VT in VITAL: a study in model construction and knowledge reuse. *International Journal of Human-Computer Studies*, 44,3/4:333–371, 1996.
18. 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.
19. 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.
20. Object Management Group (OMG). XMI Specification. *www.omg.org*, 1999.
21. H. Peltonen, T. M nnist , 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.
22. J.E. Robbins, N. Medvidovic, D.F. Redmiles, and D.S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *20th Intl. Conference on Software Engineering*, pages 209–218, Kyoto, Japan, 1998.
23. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
24. 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.
25. T. Soininen, J. Tiihonen, T. M nnist , and R. Sulonen. Towards a General Ontology of Configuration. *AIEDAM, Special Issue: Configuration Design*, 12,4:357–372, 1998.
26. M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2), June, 1997.
27. W3C. Extensible Markup Language (XML). *www.w3.org*, 1999.
28. 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.