Contextual Diagrams as Structuring Mechanisms for Designing Configuration Knowledge Bases in UML

Alexander Felfernig, 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.

Abstract. Lower prices, shorter product cycles, and the customer individual production of highly variant products are the main reasons for the success of product configuration systems in various application domains (telecommunication industry, automotive industry, computer industry). In this paper we show how to employ UML in order to design complex configuration knowledge bases. We introduce the notion of contextual diagrams in order to cope with the intrinsic complexity of configuration knowledge. Since domain experts mostly think in terms of contexts, this approach leads to a more intuitive way of modeling configuration knowledge.

1 Introduction

The mass customization paradigm [18] created big challenges for the development of software supporting the product development process. A successful approach to master these challenges is to employ knowledge-based systems with domain specific, high level, formal description languages which allow a clear separation between domain knowledge and inference knowledge. Especially product configuration systems are increasingly applied for supporting product development processes. However, these systems are not integrated in the industrial software development process and their representation formalisms are not understandable for domain experts. In order to meet these challenges we propose the employment of the Unified Modeling Language (UML [20]) as domain specific notation for a conceptual design of configuration knowledge bases which can automatically be translated into the representation formalism of the corresponding configurator. The application of UML is motivated by the wide spread use of the language, the high degree of understandability, the extendability for domain specific purposes, and the availability of a build-in constraint language which allows the definition of complex constraints.

A configuration task can be characterized through a set of components, a description of their properties (attributes and their domains), available connection points (ports), and constraints on legal configurations. Given some customer requirements, the result of computing a configuration is a set of components,

© Springer-Verlag Berlin Heidelberg 2000

corresponding attribute valuations, and connections satisfying all constraints and the customer requirements. In order to enable the construction of product models in UML we have defined a profile which contains domain-specific modeling concepts including the corresponding well-formedness rules. Furthermore, we have defined a set of translation rules which allow the automatic translation of the conceptual representation into an executable logic representation [8], [9] based on the component port model [16], [11].

The proposed development process for configuration knowledge bases is shown in Figure 1. First, a conceptual model of the configurable product is designed in UML (1). After syntactic checks on the correct usage of the concepts, the resulting model is non-ambiguously translated into logical sentences (2) which can be exploited by a general configuration engine for computing product configurations. The resulting knowledge base is validated by the domain expert using test runs on examples (3). A valid configuration knowledge base is employed in productive use (4).

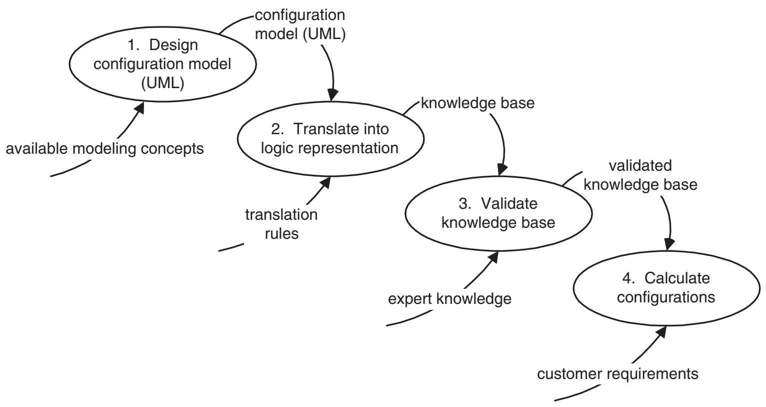
1. Design configuration model (UML)
configuration model (UML)
available modeling concepts
2. Translate into logic representation
knowledge base
translation rules
3. Validate knowledge base
validated knowledge base
expert knowledge
4. Calculate configurations
customer requirements

Fig. 1. Configuration system development process

Designing configuration knowledge bases using a conceptual modeling language combined with the automatic translation into an executable logic representation results in significant reductions of development efforts. However, when applying our approach for representing complex configuration knowledge bases, we have made the experience that the structuring mechanisms provided by UML do not suffice. In order to improve the understandability and maintainability of configuration models we introduce the notion of contextual diagrams[1], which provide a means for structuring the domain knowledge in a more intuitive way.

The paper is organized as follows. After giving a simple example for modeling configuration knowledge bases using UML (Section 2) we briefly discuss the properties of a configuration task and give a formal definition which is further used as basis for translating UML models (Section 3). In the following we motivate the usage of contextual diagrams by showing the limits of conventional

[1] This notion of context must not be confused with an OCL context.

structuring mechanisms provided by UML. Furthermore, we give an example for the construction of contextual diagrams (Section 4). In Section 5 we show how to translate contextual diagrams into a logic representation discussed in Section 3. Finally, Sections 6 and 7 contain related work and conclusions.

2 Building Configuration Models Using UML

Figure 2 shows how a configurable product can be modeled using an UML class diagram. Such a diagram describes 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. Figure 2 shows how UML can be employed for modeling configurable products using the built-in extension mechanisms (stereotypes). A configuration model created with the defined concepts can automatically be translated into an executable logic representation[2].

For presentation purposes we introduce a simplified model of a configurable car as working example. We use standard UML concepts as well as newly introduced domain-specific stereotypes, whereby their usage is restricted through OCL constraints (Object Constraint Language) in the UML metamodel. The basic structure of the product is modeled using classes, generalization and aggregation of the well-defined parts (component-types) the final product can consist of. The applicability of these object-oriented concepts for configuration problems has been shown in [17]. The following modeling concepts (represented as stereotypes) are typical for the product configuration domain [22]*: component types*, *resources*, and *ports* are stereotyped classes, *connections* and *compatibility constraints* are stereotyped relations, *requires*, *produces*, and *consumes* are stereotyped dependencies.

- **Component types** Component types represent parts the final product can be built of. They are characterized by attributes (e.g. the component type *lights* is characterized by the attribute *color*).
- **Generalization** Component types with a similar structure are arranged in a generalization hierarchy[3].
- **Aggregation** Aggregations between components represented by part-of structures state a range of how many subparts an aggregate can consist of. Within the configuration domain the part of concept has the following semantics. A component is either a compositional part of another component, i.e. it must not be part of another component, or it is a non-composite (shared) part, i.e. the component can be shared between different components.
- **Resources, produces, and consumes** Parts of a configuration problem can be seen as a resource balancing task, where some of the component types *produce* some resource and others *consume* a *resource* (e.g. a *radio* represents a consumer of the resource *battery*).

[2] A detailed discussion on the translation rules is given in [8], [9].

- **Ports and connections** 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. Ports represent connection points between connected components. E.g. a *radio power supply* must be connected to the *battery*.
- **Compatibility and requirements relations** Some types of components cannot be used in the same final configuration - they are *incompatible* (e.g. a *4-wheel gearing* is incompatible with a *diesel engine*). In other cases, the existence of one component type *requires* the existence of another special type in the configuration (e.g. an *automatic gearing* requires an *otto engine*).
- **Additional modeling concepts and constraints** Constraints on the product model, which can not be expressed graphically, are formulated using the language OCL (Object Constraint Language), 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[4]. The discussed modeling concepts have shown to cover a wide range of application areas for configuration [17]. 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.

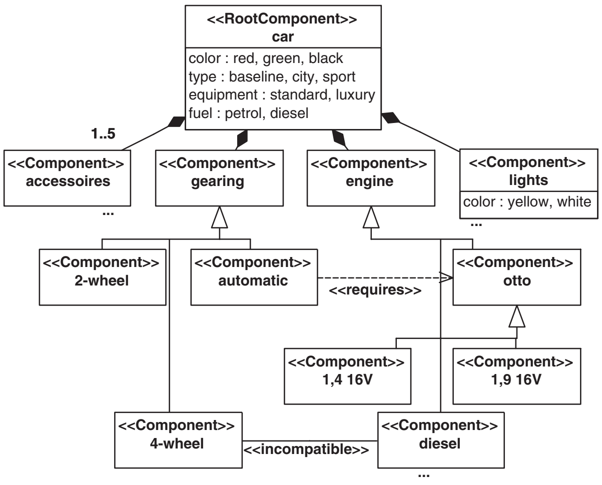

Fig. 2. Simple configuration model of a car

[3] Note that the car type in Figure 2 could also be represented in a generalization hierarchy.

[4] A detailed discussion on the translation of OCL expressions can be found in [9].

After having defined the configurable product, the actual configuration can take place. The user (the customer) specifies the requirements for the actual variant of the product. Let {*color:red, type:city, equipment: standard, engine:diesel*} be the customer requirements. The configuration system selects a *2-wheel* gearing since a *diesel* engine is incompatible with a *4-wheel* gearing as well as with an *automatic gearing* (*automatic* gearing requires an *otto engine*). Furthermore, a set of *yellow lights* and one *accessoire* (not further refined in this example) are added to the configuration.

3 Configuration Task

The following definition of a configuration task is based on a consistency-based approach. A configuration problem can be seen as a logic theory that describes a component library, a set of constraints, and customer requirements. Components are described by attributes and ports. Ports are used as connection points between components. The result of a configuration task is a set of components, their attribute values, and connections that satisfy the logic theory. This model has proven to be simple and powerful to describe general configuration problems and serves as a basis for configuration systems as well as for representing technical systems in general [16], [24], [14].

The formulation of a configuration problem can be based on two sets of logic sentences, namely *DD* (domain description) and *SRS* (System Requirements Specification). We restrict the form of the logical sentences to a subset of range restricted first-order-logic with a set extension and interpreted function symbols. In order to assure decidability, we restrict the term-depth to a fixed number.

DD includes the description of the different component types (*types*), named ports (*ports*), and attributes (*attributes*) with their domains (*dom*).

The *DD* of the car configuration model of Figure 2 is represented as follows:

```
types = {car, accessoires, gearing, engine, 4-wheel, automatic, 2-wheel, otto,
                    diesel, 1,4 16V, 1,9 16V, lights}.
attributes(car) = {color, type, equipment, fuel}. ...
dom(car, color)= {red, green, black}. ...
ports(car) = {accessoires-port, gearing-port, engine-port, lights-port}5.
ports(engine) = {car-port}. ...
```

Additionally, constraints are included, reducing the possibilities of allowed combinations of components, connections and value instantiations.

SRS includes the user requirements on the product which should be configured. These user requirements are the input for the concrete configuration task.

The configuration result is described through sets of logical sentences (*COMPS*, *ATTRS*, and *CONNS*). In these sets, the employed components, the attribute values (parameters), and the established connections are represented.

[5] The *part of* relationship between components is translated into connections between component ports in the logical representation.

– *COMPS* is a set of literals of the form *type(c,t)*. *t* is included in the set of types defined in *DD*. The constant *c* represents the identification for a component.
– *CONNS* is a set of literals of the form c*onn(c1,p1,c2,p2)*. *c1* and c2 are component identifications from *COMPS*, *p1* (*p2*) is a port of the component *c1 (c2)*.
– *ATTRS* is a set of literals of the form *val(c,a,v)*, where c is a component-identification, a is an attribute of that component, and v is the actual value of the attribute (selected out of the domain of the attribute).

Example for a configuration result:

```
type(c1, car).
type (a1, accessoires).
type(g1, 2-wheel).
type(e1, diesel).
type(l1, lights).
conn(c1, accessoires-port, a1, car-port).
conn(c1, gearing-port, g1, car-port).
conn(c1, engine-port, e1, car-port).
conn(c1, lights-port, l1, car-port).
```

Based on these definitions, we are able to specify precisely the concept of a consistent configuration:

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

Additionally we have to specify that *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 configuration.*

This is accomplished by additional logical sentences which can be generated using *DD*, *COMPS*, *CONNS*, and *ATTRS*. A configuration, which is consistent and complete w.r.t. the domain description and the customer requirements, is called a *valid configuration.* A detailed formal exposition is given in [11].

Using this component port formalism the following sentence can be derived expressing the *requires* relation between *automatic* gearing and *otto engine*[6]:

$$\mathsf{type(ID1, automatic)} \land \mathsf{type(ID2, car)} \land \mathsf{conn(ID1, car\text{-}port, ID2, gearing\text{-}port)} \Rightarrow \\ \exists\ \mathsf{ID3\ type(ID3, otto)} \land \mathsf{conn(ID3, car\text{-}port, ID2, engine\text{-}port)}.$$

The *incompatible* relation between a *4-wheel* gearing and a *diesel engine* can be translated as follows:

$$\mathsf{type(ID1, 4\text{-}wheel)} \land \mathsf{type(ID2, diesel)} \land \mathsf{conn(ID1, car\text{-}port, ID3, gearing\text{-}port)} \land \\ \mathsf{type(ID3, car)} \land \mathsf{conn(ID2, car\text{-}port, ID3, engine\text{-}port)} \Rightarrow \mathsf{false}.$$

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

The reason for applying the component port model as formal representation of UML configuration models lies in the direct executability of this representation in various configuration systems (e.g. [14], [24]).

In addition to the constraints derived from the UML model of Figure 2 further constraints are derived, e.g. *if component X is connected to component Y then component Y is connected to component X too,* or *one port can be connected to exactly one other port*, or *components have a unique type.* These constraints are called application independent constraints, which can automatically be generated from the domain description *(DD).*

In order to express constraints not representable graphically (using the concepts presented in Section 2), OCL constraints can be employed. These constraints often concern different model classes sometimes stored in different packages. The maintenance of these constraints becomes difficult, even if only a small set of constraints is formulated in the model (see Figure 3). The definition and maintenance of textually represented constraints is a real challenge for the knowledge engineer as well as the technical expert, whereas graphical representations significantly enhance the flexibility regarding human computer interaction and consequently reduce development and maintenance costs. In order to improve knowledge acquisition of such constraints we introduce the notion of contextual diagrams which will be presented in the next section. The idea is to divide the configuration model into different contexts in which different constraints must hold, e.g. *if the customer selects a car of type baseline, the gearing must be of type automatic* (see constraint C1 in Figure 3). The expression *gearing must be of type automatic* must hold, if a car of type *baseline* is selected, i.e. it must only hold in the context *car is of type baseline.* These kinds of constraints can be expressed graphically by their specification in a separate diagram denoted as contextual diagram, which is itself an UML class diagram. The argument for the application of contextual diagrams lies in the user centered knowledge acquisition support and in the reduction of textual constraints in the model. In order to represent the dependency between different contextual diagrams a diagram hierarchy must be introduced.

In the following sections we show how to employ contextual diagrams as structuring mechanism for modeling complex (configuration) knowledge bases.

4 Building Contextual Diagrams

UML packages and views primarily support the grouping of model elements not regarding the structure of complex constraints as shown in Figure 3. In order to tackle this challenge we propose the notion of contextual diagrams (representing contexts[7]), which allow the organization of (configuration) knowledge in a more intuitive way. The notion of context has been discussed in different research areas [4], [15], [21], [3], [25] using quite different interpretations. An overview of different interpretations of the notion of context can be found in [3].

[7] We assume that contextual diagrams and the corresponding contexts are denoted by the same name.

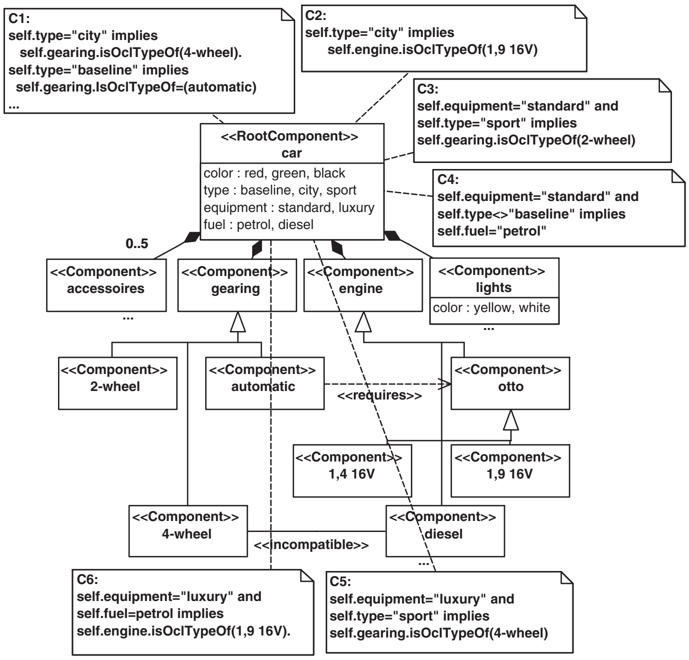
C1:
self.type="city" implies
self.gearing.isOclTypeOf(4-wheel).
self.type="baseline" implies
self.gearing.IsOclTypeOf=(automatic)
...
C2:
self.type="city" implies
self.engine.isOclTypeOf(1,9 16V)
C3:
self.equipment="standard" and
self.type="sport" implies
self.gearing.isOclTypeOf(2-wheel)
C4:
self.equipment="standard" and
self.type<>"baseline" implies
self.fuel="petrol"
<<RootComponent>>
car
color : red, green, black
type : baseline, city, sport
equipment : standard, luxury
fuel : petrol, diesel
0..5
<<Component>>
accessoires
...
<<Component>>
gearing
<<Component>>
engine
<<Component>>
lights
color : yellow, white
...
<<Component>>
2-wheel
<<Component>>
automatic
<<requires>>
<<Component>>
otto
<<Component>>
1,4 16V
<<Component>>
1,9 16V
<<Component>>
4-wheel
<<incompatible>>
<<Component>>
diesel
...
C6:
self.equipment="luxury" and
self.fuel=petrol implies
self.engine.isOclTypeOf(1,9 16V).
C5:
self.equipment="luxury" and
self.type="sport" implies
self.gearing.isOclTypeOf(4-wheel)

Fig. 3. Car model with additional sales constraints (defined in OCL)

Our interpretation of context is similar the the notion of context defined by McCarthy [15], who proposes a formalism *ist(c,p)* in order to define a context *c*, in which the proposition *p* must hold, i.e. *p is true* in *c*.

Definition (context) A context *c* is a tuple *(Prec, Expr)*, which is to be taken as assertion that *Expr* is *true* if *Prec* holds, i.e. *Prec⇒Expr*. *Prec* denotes a conjunction of logical expressions $Prec_i$, *Expr* either denotes a set of logical expressions $Expr_i$, or denotes a further context c' represented by the tuple *(Prec', Expr')*.

Following this definition we are able to construct a hierarchy of contextual diagrams (UML class diagrams), in which each contextual diagram c_i either represents the root of the hierarchy, or is derived from a contextual diagram c_j ($c_i \neq c_j$); Figure 4 shows a simple context hierarchy consisting of the root context c_{root} and the derived contexts $c_{baseline}$, c_{sport}, and c_{city}. *Prec* and *Expr* of the context $c_{baseline}$ can be expressed as OCL constraints as follows[8].

$c_{baseline}= \{ \underbrace{(car.type =' baseline'}_{Prec_1},$

[8] Note, that in many cases (also in this example) the designer of the configuration model does not have to formulate the constraints textually.

$$\underbrace{(car.accessoires \rightarrow size \succeq 0 \; and \; car.accessoires \rightarrow size \leq 2))}_{Expr_1},$$
$$\underbrace{gearing.isOclType(automatic)}_{Expr_2},$$
$$\underbrace{engine.isOclType(otto)}_{Expr_3}\}$$

The constraints given in the example can be entered graphically as shown in Figure 5. First, the designer simply formulates the precondition of the new context c_i - in this case by reducing the domain of the attribute *type*, i.e. *type: baseline*. After having defined the precondition *(Prec)* of the new context c_i, the designer enters a set of constraints *(Expr)* for c_i - in this case the domain of the engine type is reduced to *otto* (selection of type *otto*), the gearing type to *automatic* (selection of type *automatic*), and the multiplicity of the part of relationship between *car* and *accessoires* to *0..2* (upper bound of multiplicity reduced to 2).

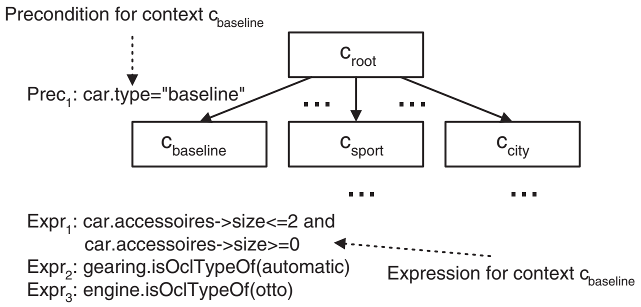
Precondition for context c_baseline
Prec1: car.type="baseline"
c_root
c_baseline
c_sport
c_city
Expr1: car.accessoires->size<=2 and car.accessoires->size>=0
Expr2: gearing.isOclTypeOf(automatic)
Expr3: engine.isOclTypeOf(otto)
Expression for context c_baseline

Fig. 4. A simple context hierarchy

Assuming that the contextual diagram c_{root} exists, a contextual diagram c_i ($c_i \neq c_{root}$) is constructed as follows (see Figure 6).

1. Select an existing contextual diagram c_j.
2. Define a precondition *Prec*$_i$ for c_i. Preconditions can either be defined graphically by specialization of classes of c_j, by restriction of the multiplicity of classes of c_j, or restriction of attribute domains of c_j. Furthermore, a precondition can be formulated as OCL expression.
3. Add the additional expressions *Expr*$_i$ valid in c_i by specialization of classes of c_j, by restriction of the multiplicity of classes of c_j, by restriction of attribute domains of c_j, or by using the following concepts presented in Section 2 *(requires, incompatible, produces, consumes, connected)*. Furthermore, *Expr*$_i$ can be formulated as OCL expression.
4. Derive the new contextual diagram c_i, i.e. represent only the selected (specialized) classes, relevant multiplicities, attribute domains, and constraints (also including additionally added constraints).

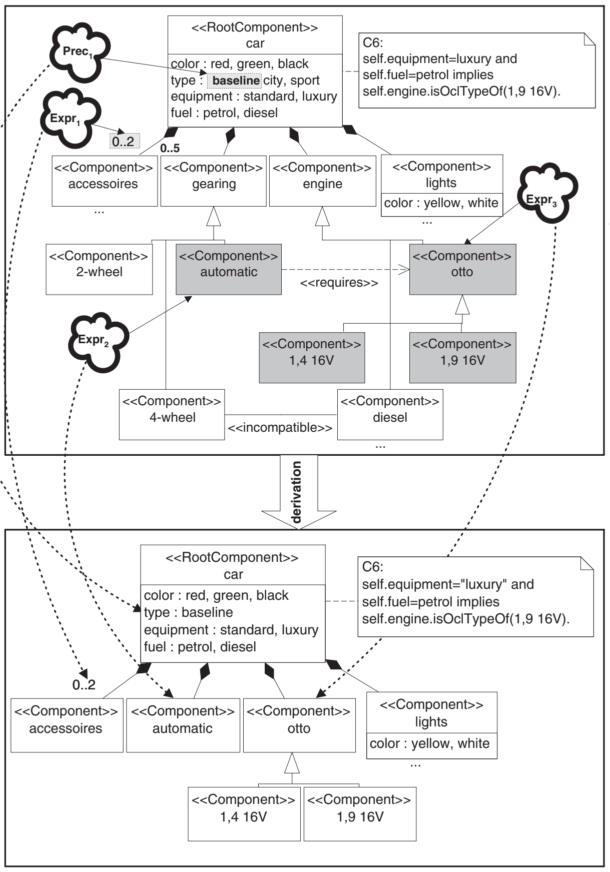

Fig. 5. Building the context $c_{baseline}$

Note, that a contextual diagram can contain only classes, attributes, generalization hierarchies, part of relations, and multiplicities, which were already defined in the diagram of the root context, i.e. no additional elements can be added in the context c_i ($c_i \neq c_{root}$). We impose this restriction since our goal is the structuring of constraints belonging to a given component hierarchy, which is defined in the *root* context.

Following the above rules we are able to construct a contextual diagram for the context $c_{baseline}$ as follows.

1. The existing contextual diagram c_{root} is selected.
2. A precondition *(Prec)* for the new context $c_{baseline}$ is formulated. In this example this is done by reducing the domain of the attribute *type* to *baseline* (see Figure 5).
3. The designer defines a set of restrictions *(Expr)* for $c_{baseline}$, namely at most two *accessoires* (not further refined in this example) can be part of the

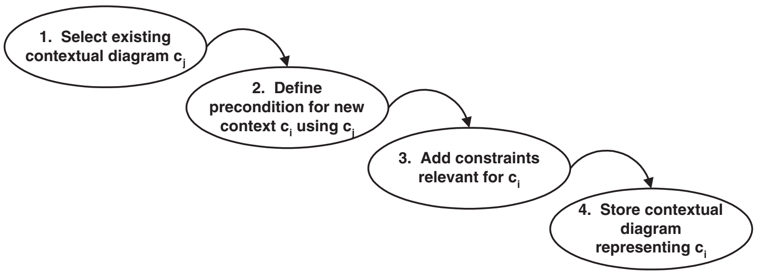
1. Select existing contextual diagram c_j
2. Define precondition for new context c_i using c_j
3. Add constraints relevant for c_i
4. Store contextual diagram representing c_i

Fig. 6. Context building process

configuration, the *gearing* must be of type *automatic*, and the *engine* unit must be of type *otto.*

4. The new contextual diagram c_i can be derived (see Figure 5).

We assume that graphically entered expressions of *(Prec, Expr)* are internally stored as OCL constraints, which are further used in order to derive a component port representation of the contextual diagram. Figure 4 shows the entered selections represented as OCL constraints.

After having defined *Prec* and *Expr*, a corresponding contextual diagram $c_{baseline}$ can be derived. This diagram can be used for the construction of further contextual diagrams inside $c_{baseline}$.

5 Translating Contextual Diagrams into Logic

In order to translate contextual diagrams into our logic representation (component port representation discussed in Section 3), the following rules must be considered:

1. Let c_{root} be the root contextual diagram representing our configuration model. Then *DD* of c_{root} is derived conforming the translation rules given in [8], [9], i.e. c_{root} is translated into the component port represention.
2. All additional expressions added in a contextual diagram c_i ($c_i \neq c_{root}$) are translated as follows. Let *Prec* be a conjunction of preconditions defined for c_i, and *Expr* a set of expressions defined for c_i. Then c_i is translated as follows: *Prec* $\Rightarrow$ *Expr,* i.e. $\forall$ $Expr_i \in Expr$ $(Prec \Rightarrow Expr_i)$ is added to *DD.* Furthermore, if c_i is a subcontext of a context c_j with preconditions *Prec'*, then c_i is translated as follows: $Prec' \Rightarrow (Prec \Rightarrow Expr)$.

Example: translation of the context $c_{baseline}$

$Prec_1$: *car.type="baseline"*
$Expr_1$: car.accessoires$\rightarrow$size$\leq$2 and car.accessoires$\rightarrow$size$\geq$0
$Expr_2$: gearing.isOclTypeOf(automatic)
$Expr_3$: engine.isOclTypeOf(otto)

Conform to $c_{baseline}$, i.e. $\text{Prec}_1 \Rightarrow (\text{Expr}_1 \wedge \text{Expr}_2 \wedge \text{Expr}_3)$ must be translated into the component port representation. An alternative representation of the implication is the following: $(\neg\text{Prec}_1 \vee \text{Expr}_1) \wedge (\neg\text{Prec}_1 \vee \text{Expr}_2) \wedge (\neg\text{Prec}_1 \vee \text{Expr}_3)$, i.e. $(\text{Prec}_1 \Rightarrow \text{Expr}_1) \wedge (\text{Prec}_1 \Rightarrow \text{Expr}_2) \wedge (\text{Prec}_1 \Rightarrow \text{Expr}_3)$. The component port representation for these constraints valid in $c_{baseline}$ is the following:

type(X, car) $\wedge$ val(X, type, baseline) $\Rightarrow$
parts(X, accessoires, Count) $\wedge$ (Count $\leq$ 2) $\wedge$ (Count $\geq$ 0).

type(X, car) $\wedge$ val(X, type, baseline) $\wedge$ type(Y, gearing) $\wedge$
conn(X, gearing-port, Y, car-port) $\Rightarrow$ type(Y,automatic).

type(X, car) $\wedge$ val(X, type, baseline) $\wedge$ type(Y, engine) $\wedge$
conn(X, engine-port, Y, car-port) $\Rightarrow$ type(Y, otto).

Note, that the generated sentences belong to one single *DD*. The automatic generation of the knowledge base is done by traversing the derived context hierarchy, translating the contextual diagrams into the component port representation, and adding the result to *DD*.

In order to support the debbugging of knowledge bases we propose a consistency-based formalism (model-based diagnosis) discussed in [10], where the constraints in the generated configuration knowledge base are interpreted as components which are diagnosed in order to find contradicting components. The resulting diagnoses are presented to the user indicating the corresponding classes in the conceptual configuration model.

6 Related Work

The formalization of the semantics of conceptual modeling languages like OMT [19], UML [20] and OCL is an actual research area [2], [5], [6]. These formalizations are mostly based on mathematical models and specification languages. The work is focused on getting precise definitions of the employed concepts for general models. In order to support the graphical representation of complex constraints [12] propose the notion of constraint diagrams, which are basically extensions of Venn Diagrams. We view our work as complementary, since our goal is to generate formal descriptions which can be interpreted by logic-based problem solvers restricted to a special domain (solvers based on the logic representation discussed in Section 3). However, the consideration of the concepts presented in [12] could be useful for our work in order to provide additional modeling concepts for configuration domains.

There is a long history in developing configuration tools in knowledge-based systems (see [23]). However, the automated generation of logic-based knowledge bases by exploiting a formal definition of standard design descriptions like UML has not been discussed so far. Comparable research has been done in the fields of automated and knowledge-based Software Engineering, e.g. the derivation of programs in the Amphion [13] project.

Research dealing with knowledge representation in the configuration domain regards declarative constraint representation as the basic representation formalism, since the development and maintenance of rule-based systems like R1/XCON [1] have shown to be error-prone. Using declarative constraint representation is not enough since knowledge bases become increasingly complex so that conventional structuring mechanisms do not suffice. Conventional mechanisms for structuring knowledge bases in the configuration domain are described in [7]. The notion of context has been discussed in several research areas using different interpretations depending on the application area. In [25] a context is denoted as a higher order conceptual entity, which describes a group of conceptual entities from a particular point of view. Rules and operations are provided to organize the manipulation of contexts. Furthermore, an example for the application of contexts in a cooperative document design environment is given, where documents represent collections of objects. In [4] the notion of context is employed for supporting cooperative work in hypermedia design providing a set of context operations on hypermedial objects (e.g. editing, inquiry, attribute operations). Compared to these approaches we view our approach as complementary, since our goal is to effectively support the knowledge acquisition for complex (configuration) knowledge bases not regarding any operations for combining different contexts (all contexts are translated into the same *DD*). In AI research the notion of context is interpreted in a different sense. [15] denotes a context as an abstract object representing a particular meaning not true outside the context. Our approach applies these concepts in order to support knowledge acquisition for complex knowledge bases on a conceptual (UML) level, especially improving the representation of complex constraints in order to be understandable for domain experts. Furthermore, an overview of different interpretations of context in the design area is given in [3].

7 Conclusions

Extensible standard design languages like UML are able to provide a basis for introducing and applying rigorous formal descriptions of application domains. This approach helps us to combine the advantages of various areas. First, high level formal description languages reduce the development time and effort significantly, because these descriptions are directly executable. Second, standard design languages like UML are far more comprehensible and are widely adopted in the established industrial software development process. We defined a logic-based formal semantics for UML constructs, which allows us to generate logical sentences and to process them by a problem solver. The intrinsic complexity of configuration knowledge requires the provision of concepts which effictively support the representation of complex constraints, i.e. support a graphical representation. We have shown how to represent and organize complex (configuration) knowledge bases using the concept of contextual diagrams which especially support an understandable representation of complex constraints.

Applying these concepts enables us to automate the generation of specialized software applications and allows for rapid generation of prototypes. An improvement in the requirements engineering phase through short feedback cycles is achieved. The design model is comprehensible for domain experts and can be adapted and validated without the need of specialists. Consequently, time and costs for the development and maintenance of product configuration systems can be reduced significantly. We chose product configuration systems because of the economic and technical relevance and the challenges to be tackled by software engineering of these 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. 252
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. 251
3. Charles Charlton and Ken Wallace. Reminding and context in design. In *Proceedings 6th International Conference on Artificial Intelligence in Design (AID'00)*, pages 569–588, Boston/Worcester, MA, USA, 2000. Kluwer Academic Publishers. 246, 246, 252
4. N.M. Delisle and M.D. Schwartz. Contexts – a partitioning concept for hypertext. *ACM Transactions on Information Systems*, 5,2:168–186, 1987. 246, 252
5. A. Evans. Reasoning with UML class diagrams. In *Proceedings Workshop on Industrial Strength Formal Methods(WIFT'98)*, Florida, USA, 1998. IEEE Press. 251
6. A. Evans and S. Kent. Core Meta-Modeling Semantics of UML: the pUML aproach. In *Proceedings ¡¡UML¿¿'99*, pages 140–155, Fort Collings, Colorado, USA, 1999. 251
7. F. Feldkamp, M. Heinrich, and K.D. Meyer Gramann. SyDeR System Development For Reusability. *AIEDAM, Special Issue: Configuration Design*, 12,4:373–382, 1998. 252
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. 241, 242, 245, 250
9. A. Felfernig, G. Friedrich, and D. Jannach. Generating product configuration knowledge bases from precise domain extended UML models. In *Proc. 12th International Conference on Software Engineering and Knowledge Engineering*, pages 284–293, Chicago, USA, 2000. 241, 242, 243, 250
10. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. An Integrated Development Environment for the Design and Maintenance of Large Configuration Knowledge Bases. *Artificial Intelligence in Design (AID'00) (to appear), Kluwer Academic Publisher*, 2000. 251
11. G. Friedrich and M. Stumptner. Consistency-Based Configuration. In *AAAI Workshop on Configuration, Technical Report WS-99-05*, pages 35–40, Orlando, Florida, 1999. 241, 245
12. J. Gil, J. Howse, and S. Kent. Constraint Diagrams: A Step Beyond UML. In *Proceedings TOOLS USA'99*. IEEE Press, 1999. 251, 251
13. M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proc. 9th Knowledge-Based Software Engineering Conference*, pages 48–57, Montery, CA, USA, 1994. IEEE Computer Society. 251
14. D. Mailharro. A classification and constraint-based framework for configuration. *AIEDAM, Special Issue: Configuration Design*, 12,4:383–397, 1998. 244, 246

15. J. McCarthy. Notes on formalizing context. In *Proc. of the 13th IJCAI*, pages 555–560, Chambery, France, 1993. 246, 247, 252
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. 241, 244
17. H. Peltonen, T. Mnnist, 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. 242, 243
18. B.J. PineII, B. Victor, and A.C. Boynton. Making Mass Customization Work. *Harvard Business Review*, Sep./Oct. 1993:109–119, 1993. 240
19. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W.Lorensen. Object-Oriented Modeling and Design. In *Prentice Hall International Editions*, New Jersey, USA, 1991. 251
20. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998. 240, 251
21. M. Siegel, E. Sciore, and S. Salveter. A method for automatic rule derivation to support semantic query optimization. *ACM Transactions on Database Systems*, 17:563–600, 1992. 246
22. T. Soininen, J. Tiihonen, T. Mnnist, and R. Sulonen. Towards a General Ontology of Configuration. *AIEDAM, Special Issue: Configuration Design*, 12,4:357–372, 1998. 242
23. M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2), June, 1997. 251
24. M. Stumptner, G. Friedrich, and A. Haselbck. Generative constraint-based configuration of large technical systems. *AIEDAM, Special Issue: Configuration Design*, 12, 4:307–320, Sep. 1998. 244, 246
25. Manos Theodorakis, Anastasia Analyti, Panos Constantopoulos, and Nikos Spyratos. Context in information bases. In *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS'98)*, pages 260–270, New York City, NY, USA, August 1998. IEEE Computer Society. 246, 252