Generating product configuration knowledge bases from precise domain extended UML models

Alexander Felfernig, Gerhard E. Friedrich, and Dietmar Jannach
Institut für Wirtschaftsinformatik und Anwendungssysteme
University of Klagenfurt
9020 Klagenfurt, Austria
{felfernig,friedrich,jannach}@ifi.uni-klu.ac.at

Abstract

The Unified Modeling Language (UML) is an emerging standard conceptual modeling language in Software Engineering processes. UML provides extension mechanisms (stereotypes) to adapt the general modeling language to specific application domains. In addition, UML comprises the standardized expression language Object Constraint Language (OCL) to model additional invariants.

In this paper we show how UML can be extended using the standardized extension mechanisms to fit the needs of the domain of knowledge-based product configuration. Starting from a conceptual product model, we define the semantics of the employed extensions based upon a logic theory of configuration in a way that the resulting knowledge bases can be processed by a specialized inference engine. We show especially how the built-in expression language OCL can be employed to enhance the expressiveness of the conceptual models and how such expressions can be translated to the logical model. Finally, we describe a prototype implementation of the presented ideas based on commercial tools.

1 INTRODUCTION

According to the paradigm of mass-customization, products are nowadays sold to customers in many variants according to the specific customer demands. Product configuration systems[1] (or *configurators* for short) have been one of the most successful applications of AI technology for the last two decades ([7],[21]), e.g., in the automotive and telecommunication industries. They support the technical engineer, the sales representative or the customer to cope with the large number of variants offered and help them to configure products that both fulfill the customer's requirements and technical and economic constraints on allowed product configurations.

[1] Product configuration should not be confused with *Software Configuration Management* despite some similarities concerning components and ports.

Starting from its rule-based origins [14] configuration technology has progressed to other representation and solving mechanisms, e.g., constraint based, case based etc. For an overview of configuration technology see e.g., [21]. Despite these advances, knowledge acquisition and maintenance of configuration knowledge bases is still a critical factor since the development time is strictly limited as the configurable product and the configurator software have to be developed in parallel. Additionally, these knowledge based systems often use proprietary representation mechanisms and conceptual models which are not integrated into standard software development processes.

On the other hand, from a software engineering point of view, different conceptual modeling languages and techniques have been developed, e.g., Object Modeling Technique (OMT), Unified Modeling Language (UML) [8], to improve the software development process. These graphical notations are used to make abstractions from the real world thus making the problem domain more comprehensible and easier to communicate between the people involved. These techniques are widely accepted and employed in industrial software engineering processes.

Since these methods are general purpose modeling languages and abstract from the real world they face the problem that these models are not directly executable. In addition, the semantics of the different modeling concepts are mostly stated in plain English thus making an automated translation of the conceptual model to an executable representation impossible. *UML* (Unified Modeling Language) leverages this problem of generality of the language to some extent allowing the language to be extended by additional modeling concepts (*stereotypes*) for certain domains, e.g., for Business Process Modeling. In addition, UML comprises a declarative object oriented expression language *OCL* (Object Constraint Language) to express additional invariants on the models (which can not be expressed graphically). The applicability of an object oriented approach to model configurable products has been shown in [17] and [18]. In our previous work [5] we have shown – based on a logical theory of configuration – how UML can be employed to model configuration

problems and how these conceptual models can be automatically translated into an executable representation. In this paper we show how these models can be enhanced with additional constraints formulated in OCL and how these invariants are translated into our logic theory of configuration. This translation defines the semantics for the individual constructs of OCL precisely. In addition, we demonstrate how the presented domain specific extensions are properly added to UML using the built-in mechanisms. Finally, we describe a prototypical implementation of our approach using commercial tools. The paper ends with related work and conclusions.

2 USING UML AS NOTATION FOR CONFIGURATOR DEVELOPMENT

We will first shortly summarize how UML can be used to automatically construct configurator knowledge bases.

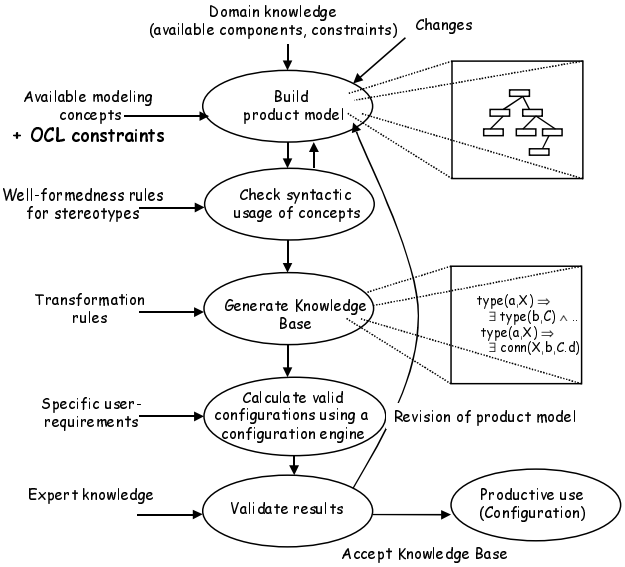


Figure 1 Configurator development process

Figure 1 shows the envisioned development process for configurators from [5]. Starting from a conceptual product model, a knowledge is generated which is executable by a specialized inference engine. The translation is done after a syntactic check and is based on precisely defined semantics of the available modeling concepts. The resulting knowledge base can be validated and accepted or it can be revised again on the conceptual level. Once the generated knowledge base is assumed to be correct, actual configurations can be calculated given some specific user requirements. In this paper we show how additional invariants that are not graphically expressible and are formulated in OCL can be employed in the product model and enrich the expressiveness of our conceptual language.

Figure 2 shows a small fragment from a conceptual product model using a UML static structure diagram with domain specific stereotypes. The basic structure of the product is modeled using part-of and generalization relations of the available component types the product can consist of.

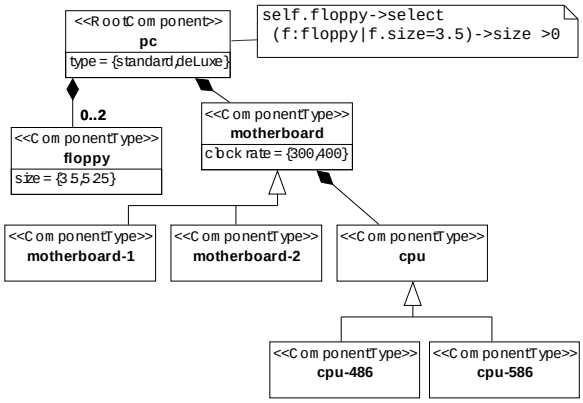


Figure 2 A conceptual model fragment using UML

Component types are described through their properties (attributes) and can be connected to other components using predefined connection slots (*ports*). In addition, typical modeling concepts for the configuration domain are introduced to express, e.g., incompatibilities between component types or overall resource constraints that must be observed by a valid configuration (for details see [5]). Figure 2 also shows an additional OCL constraint attached to the class PC, which can not be expressed graphically.

2.1 Fitting the extensions into UML

The standard way of introducing extensions to UML is to add a *profile*, which is defined to be a set of predefined stereotypes, tagged values, and constraints. A profile does not introduce any new basic concepts to the language but rather specializes it for some domain. In the configuration domain, we define stereotypes in the UML *metamodel*, e.g., rather than saying that an actual model can be built of *classes*, we define a new stereotype "*component type*" from the metamodel element *class*, such that in the actual model this new concept can be employed.

The UML profile for the configuration domain therefore consists of the following elements (not exhaustive):

Metamodel class	**Stereotype**
Model	Configuration Model
Class	Component Type
Class	Port
Class	Root Component
Class	Resource
Dependency	requires
Association	connected with
...	....

In addition, *well formedness rules* (also using OCL) are defined that describe the correct usage of the stereotypes

within a model.

Examples: ("Component Type" classes have no operations")

```
self.allOperations->size = 0
```

(A "requires" dependency can only be between two classes of stereotype "ComponentType")

```
self.client.stereotype = "ComponentType" and
self.supplier.stereotype = "Component Type".
```

Figure 3 shows how the profile for configuration is integrated into the four-layer architecture of UML. In the metamodel the extensions for the configuration domain are defined. The actual product models correspond to class diagrams and final configurations can be seen as instantiations of the model.

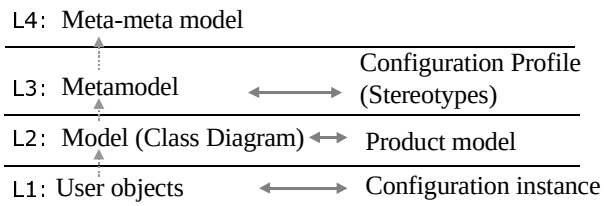


Figure 3 Layered UML language architecture

2.2 Logical model for configuration

As a target knowledge representation, into which the conceptual models are mapped with precise semantics, we defined a logical model of configuration based on the component-port model [15]. The mapping from the conceptual model to this executable representation[2] is presented in detail in [5].

We will now shortly review the basic notions of that formalism and show later on in Section 3 how OCL constraints from the conceptual model can be mapped to that representation.

Component port model

The component port model is widely accepted and used as a representation mechanism for configuration problems ([7],[13],[15]): A configurable product is built from predefined components which are characterized by a set of properties (attributes). *Ports* are used as connection points between component individuals. In order to map the product structure from the conceptual level to our logical representation we define stereotyped classes with their attributes in the UML model as component types. The *part-of* structure is mapped to our logical representation through the introduction of ports and additional constraints.

[2] We employ a logic programming notation where variable names start with an upper case letter or are written as "_". The variables are all-quantified if not explicitly mentioned.

Configuration problem

The definition of a configuration problem 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. The result of a configuration task is a set of components, their attribute values, and connections that satisfy the logic theory.

We assume the definition of the product structure to be described in a set *DD* (domain description) of logical sentences. This set contains the basic product structure and additional constraints on allowed constellations. In addition, the customer's requirements (additional constraints, initial configurations) are represented in a set *SRS* (specific requirements) of logical sentences. An individual configuration *CONF* consists of a set of components, connections and attribute values described by positive ground literals. In our examples, we employ a set of predicates *CONL={type/2, conn/4, val/3}* to describe such configurations (for special domains, additional predicates can be defined).

Definition: *(Consistent Configuration): Given a configuration problem (DD,SRS,CONL) a configuration CONF is consistent iff* $DD \cup SRS \cup CONF$ *is satisfiable.*

In addition we define the terms *valid*, *irreducible* and *complete configuration* in the detailed formal exposition [9] to come to a precise logical definition of our logical theory.

Example

The following example illustrates how the conceptual product model from Figure 2 is transformed into the domain description *DD* of our logical model of configuration.

- We define what types of components can exist in a final configuration, i.e., we add

  types = {pc,cpu,cpu-486, motherboard,...}.

  to the domain description.

- To represent the properties we add *attributes* functions, e.g.,

  attributes(floppy) = {size}.
  attributes(cpu) = {clock_rate}.

- Attributes are assigned a domain, e.g.,

  dom(floppy,size) = {3.5, 5.25}.
  dom(cpu,clock_rate) = {300,400}.

- The type hierarchy is mapped, i.e., additional constraints are added, e.g.,

  $type(ID,cpu\text{-}486) \Rightarrow type(ID,cpu).$
  $type(ID,cpu) \Rightarrow$
  $type(ID,cpu\text{-}486) \vee type(ID,cpu\text{-}586).$

After the definition of the available component types, we map the product structure given as *part-of* relations (aggregations) to the component port representation. We

therefor define ports to connect the components with its parts and aggregates. The number of needed ports is determined by the multiplicities of the aggregation. The ports are named according to the opposite class. Let the function *ports(type)* return the set of ports defined for a component type.

- Given the part-of relation between "pc" and "floppy" in the conceptual model, where an individual PC can have 1 or 2 floppies installed and a floppy can be part of exactly one PC, we extend *DD* with the correct port definitions, i.e.,

  $\{floppy\text{-}port\text{-}1, floppy\text{-}port\text{-}2\} \in ports(pc).$
  $\{pc\text{-}port\} \in ports(floppy).$

- In addition we derive a constraint stating that if there is a floppy in the configuration, it must be inserted into a PC. This property is assured using the predicate *conn(C1,P1,C2,P2)* denoting that component *C1* is connected on port *P1* with component *C2*'s port *P2*.

  $type(ID,floppy) \Rightarrow \exists (P,Port)\ type(P,pc) \wedge conn(ID,pc\text{-}port,P,Port) \wedge Port \in \{floppy\text{-}port\text{-}1,floppy\text{-}port\text{-}2\}.$

For additional typical modeling concepts for the domain of product configuration (e.g., type compatibilities, other types of part-of relations, resource balancing) the according transformation rules are defined in [5].

A valid configuration fragment *CONF* for the example given in Figure 2 can be expressed using *type*, *val*, and *conn* literals, i.e.,

CONF = { type(p1,pc). type(f1, floppy).
val(pc,type,"standard").
val(f1,size, 3.5).
conn(p1,floppy-port-1,f1,pc-port).
type(m1,motherboard-1). .... }

As we can see in Figure 2, there is an additional constraint on the number of connected floppies with a certain size defined for instances of class "pc". This constraint can not be expressed using the graphical means of UML and is therefor defined using the expression language *OCL*. In the next section we show how these OCL expressions can be used for the domain of configuration.

3 OCL CONSTRAINTS

The Object Constraint Language [22], a simple declarative expression language, serves two purposes within the UML: First, it can be used to state well-formedness rules in the metamodel as it was done in Section 2.2. Second, it can be used to make the actual models more precise, e.g., one can express further invariants in the model.

When modeling configurable products with UML there usually exist constraints on valid configurations that can not be expressed graphically. In the following we show how OCL constraints can be used to enhance the expressiveness of our product models and how they can be transformed to our logic representation. Actually, we restrict the employed constraints to use only a subset of all the features of OCL, since we only utilize features concerning static structure diagrams (not, e.g., state diagrams). Furthermore, we do not need employ OCL-features to access the metamodel, e.g., constraints on the number of *association ends*.

In addition, the features defined for methods are not relevant since we do neither have methods in our model, nor time expression. Finally, non-declarative features of OCL (iterations, *let* statement) and shorthand notations are not regarded in this context. Assuming these restrictions on the usage of OCL we are given an expression language that is applicable and of practical use for the domain of product configuration.

A typical constraint for our example, stating that *for every instance* of the class PC there must be *at any time* at least one floppy with size 3.5 installed, is expressed as:

```
context PC inv:
 pc.floppy->select(f:floppy|f.size=3.5)->size >0
```

The constraint has the following features:

- The *context* describes for which class the constraint has to hold, i.e., the constraint has to hold for all instances of the class "pc".
- A navigation expression *pc.floppy*, resulting in the collection of floppies associated with the PC. The name *"floppy"* is the default name of the association from the class "pc" to the class "floppy" in the conceptual model if it is not specified explicitly.
- A predefined collection operation *select*, resulting in a subcollection of the invoking collection consisting of elements fulfilling the same subexpression,
- access to attributes of a class (*f.size*),
- the predefined operation *size* on collections resulting in the number of elements in the collection.
- A mathematical operation ">" on integers.

Since the expressions are declarative, they can be translated to our logic based representation. We will now define the transformation rules for the OCL features when parsing an individual expression for our application domain. Note that the resulting sentence is created incrementally during the left-to-right parsing process. During that parsing process we generate a logical sentence of conjuncted predicates. We assume that the expression is syntactically correct and has been "type-checked", i.e., there are no attribute accesses to non-existent attributes, no navigations over non-existent associations or other "type" errors.

Furthermore, we need to remember information concerning the actual state of the evaluation process (e.g., actual class referenced, actual attribute, actual collection, actual subexpression) and generated variables within the predi-

cates. This variable names are used to connect the individual conjuncted predicates in the resulting sentence. Subexpressions that have still to be parsed are linked to the rest of the logical sentence using a conjunction and using the generated variable names from the actual expression.

Note that in the following we do not show the translation process on the basis of the definition of a detailed grammar for the expression language, but rather show how the individual language constructs of OCL are translated into our logic representation.

3.1 Basic data types

We assume the basic data types (e.g., *integer*, *boolean*, *real*, *string*) and constants to exist in our logical representation.

3.2 Contexts

Each OCL-expression is defined within the context of a class. Starting from this reference point (context), attributes of the class can be accessed and navigation expressions over associations can originate. This context is relevant when using the keyword "*self*" within the expression, e.g., `self.size` in the context *floppy* references the attribute *size* of an actual instance of floppy.

To have this starting point in our logical expression, we generate a fact of the form *type(C,classname)* for the resulting logical sentence that will match for all *type*-facts generated during the search for solutions, i.e., for all instances of component types. The following predicates that are generated from subexpressions and operations are conjuncted to the already generated predicates. The generated variable name *C* has to be used as a starting point.

Translation: (context) Given an OCL-expression in the context of class *classname*, we add the fact *type(Classname,classname)* to the resulting sentence.

3.3 Property access

Properties of classes in the conceptual model can be accessed in OCL expression using the "."-operator for class variables. These class variables can either refer to properties of the actual context class (using the keyword *self* - which can be omitted) or to a class which is referred to through navigation. Let *Classname* be the generated variable name of the actual parsed class on which the property access is performed.

Translation: (property access) Given an access to property *p* of class *c*, then we extend the resulting logic sentence with the conjunction of *val(Classname,p,PValue*).

Note that the variable "*PValue*" then contains the actual value of the variable, on which we can apply logical or arithmetic operators.

3.4 Basic operators and constants

We assume the basic operators ("+", "-", "or", ...) for the individual elementary datatypes to exist in our logical representation. When parsing an expression with a basic operator, the operator can be directly applied on the last referenced attribute and conjuncted with the current sentence. Constants are also directly translated.

Example (attribute access and operators):

The OCL constraint

```
context: pc inv:
       self.type = "DeLuxe"
```

will be translated to:

type(PC,pc) $\wedge$ *val(PC,type,TypeValue)* $\wedge$ *TypeValue = "DeLuxe"*.

Note, that the predicates of the resulting logical sentence are linked with conjuncts and the usage of the correct variable names.

3.5 Navigation over associations

Navigation can be done in OCL using the overloaded "."-operator for classes (e.g., `pc.floppy`). This navigation expression in OCL evaluates to a collection of instances associated with an instance of the actual class. In our models of configurable products, navigation can only take place over aggregation associations between component types and "connected with" associations between ports (see [5]).

Translation: (navigation) Given a navigation expression from an instance *C1* of class *c1* to class *c2* then we add the conjunction of the predicate *connected(C1,c2,<C2_SET>)* to our logical sentence.

In addition, we define *connected(C1,c2,<C2_SET>)* as (using a LDL-like notation [4]):

connected(C1,c2,setof(<C2_SET>)) $\Leftarrow$
type(C2_SET,c2)
$\wedge$ *conn(C2_SET,c1-port,C1,C2_Port)*
$\wedge$ *C2_Port* $\in$ *{c2-port*$_1$*, ..., c2-port*$_n$*}*

Note that the predicate *connected* collects all the instance identifiers from type facts where these instances are connected to one of the ports of *c1* reserved for connections to *c2*-instances. The variable *C2_SET* then contains all corresponding instances and can be used for further operations on that collection. All further operations on the resulting collection have to use that variable name to refer to that collection correctly. This property is ensured during the parsing process (see example below).

3.6 Basic set operations

We assume that our language for logical sentences contains the basic mathematical set operations, e.g., *union, intersection, cardinality*.

Example: size (Cardinality) of a set

```
context PC inv:
  pc.floppy->size = 1 or
  pc.floppy->size = 2.
```

is translated to:

(type(PC,pc) ∧
connected(pc,floppy,setof(<Floppy_SET>)) ∧
size(Floppy_SET) = 1)
∨
(type(PC,pc) ∧
connected(pc,floppy,setof(<Floppy_SET>)) ∧
size(Floppy_SET) = 2)

where *connected(pc,floppy,setof(<Floppy_SET>))* is defined as:

connected(PC,floppy,setof(<Floppy_SET>))⇐
type(Floppy_SET,floppy) ∧
conn(Floppy_SET,pc-port,PC,Floppy_Port) ∧
Floppy_Port∈ {floppy-port-1,floppy-port-2}.

3.7 Select and reject operation

Sometimes a navigation expression evaluates to a collection, where we are only interested in a special subset of that collection. OCL offers the *select* and *reject* operations on collections to filter out elements that fulfill some boolean expression, e.g., the fact that the collection of floppies of size "3.5" connected to a PC must be "1" is defined in OCL as

```
context PC inv:
pc.floppy->select(f:floppy|f.size = 3.5)
  ->size = 1
```

The *reject* operation results in a collection of elements, except those where the boolean expression on the elements evaluates to *true*. We will not consider it here because this can be achieved by simply negating the boolean expression.

Translation (select operation) For the translation of one *select*-operation, we define a predicate of the form

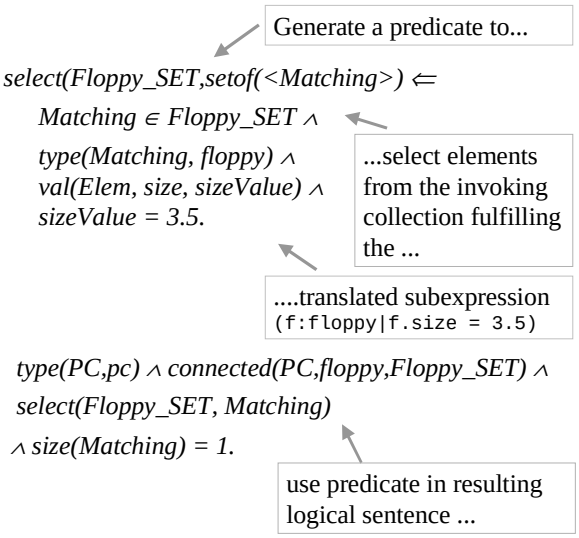
select(InvokingColl,setof(<Matching>))

using LDL-notation, where *InvokingColl* refers to the set of component identifications (which could have resulted from a navigation expression) from which some elements have to be selected and add it to the resulting logical sentence. The variable *Matching* then contains the resulting set of "matching" elements, i.e., the set of component IDs, for which the subexpression in the OCL - invariant is true. This variable can then be used in the rest of the resulting logical sentence.

In addition, we construct the definition for that *select*-predicate from the subexpression of the OCL-constraint conforming to our translation rules. To ensure that the correct component identifications are selected, we use the variable name *Matching* and add the conjunct *Matching ∈ InvokingColl* to the generated predicate definition.

Example: (Selection of 3.5-sized floppies of the PC)

Note that for every *select* and *reject* operation within an expression an individual *select*-predicate has to be derived (having a unique name).



3.8 Collect operation

Basically, the *collect* operation on collection in OCL is used to refer to a collection of attributes that is derived from a collection of objects. The syntax is the same as for the *select* operation, except that we can define what elements should be returned by the collect operation.

Example: (collect operation)

```
context pc inv:
pc.floppy->collect(f:floppy|f.size)
```

will return a collection of all attributes *size* of floppies associated with a PC.

Note that the result of the *collect* operation returns rather a *bag* than a *set*, because if there are attributes with the same value, they have to be included more than once in the result. The resulting bag of attribute values can be used in further expressions, e.g., a constraint on the sum of the values.

Translation (collect operation): In principle the translation is the same as for the *select*-operation, except that we do not construct a collection of component identifications, but a collection of attribute instances. Therefor, we construct a predicate of type

collect(InvokingColl,bagof(<Matching_Attr>))

and add it to the resulting logical sentence where *Matching_Attrs* contains a bag of attribute instantiations on which further operations can be applied. (We assume that our logical language comprises a *bag* extension).

The predicate *collect* is constructed by translating the subexpression using a *val*-predicate for all elements of the invoking collection and introducing a variable *Elem* to refer to the elements of the invoking collection (*Elem ∈ InvokingColl*) and link the translated expression on the elements using that variable name.

In the example `floppy->collect(f:floppy|f.size)`, the *collect* predicate will be defined as:

collect(Floppy_SET,bagof(<Matching_Attr>)) ⇐
Elem ∈ Floppy_SET ∧ type(Elem,floppy) ∧
val(Elem,size,Matching_Attr).

3.9 Forall operation

Using this OCL operation one can define constraints that have to hold for all elements of a collection.

Example: (All floppies must be of size "3.5")

```
context pc inv:
pc.floppy->forall(f:floppy|f.size = 3.5)
```

Again, the *forall* operation takes a subexpression as parameter, where one can introduce variables and state expressions on that variables.

Translation (forall operation) Given a collection in OCL where some additional subexpressions have to hold for all elements, we translate that subexpression according to our transformation rules and link it to the rest of our generated predicate. Therefor we generate the predicate for the subexpression on a newly introduced variable *Elem* and add a conjunct *Elem ∈ InvokingColl* to ensure that the predicate holds for all elements of the invoking collection (referred to as *InvokingColl*)

Example (All floppies must be of size "3.5".)

type(PC,pc) ∧
connected(PC,floppy,setof(<Floppy_SET>))∧
Elem ∈ Floppy_SET ∧ val(Elem,size,SizeValue) ∧
SizeValue = 3.5.

3.10 Exists operation

Using the OCL operation *exists* on collection, one can state a constraint, that at least one of the elements of a collection fulfills a given expression.

Example (there is at least one floppy with size "3.5")

```
context pc inv:
pc.floppy->exists(f:floppy|f.size = 3.5)
```

Since this can be seen as a shorthand variant of

```
pc.floppy->select
   (f:floppy|f.size = 3.5)->size > 0
```

we can translate this expression using the translation rules defined for the *select* operation and add a conjuncted predicate of the form *size(Matching) > 0* to the resulting sentence.

3.11 oclIsTypeOf

From the predefined properties on all objects in OCL we only support the predicate *oclIsTypeOf*, which can be used to decide whether a given object has a certain type (which is defined in the generalization hierarchy). The following example shows how such expressions can be used in the configuration domain.

Example: (if type of the motherboard is "motherboard-1", the clock rate must be set to "300")

```
context motherboard inv:
  self.oclIsTypeOf(motherboard-1)implies
     self.clock_rate = "300"
```

According to the definitions we made when mapping generalization hierarchies to our logical model, we have to include an additional *type* fact with the corresponding component type given in the *oclIsType* predicate parameter.

Translation (oclIsTypeOf): Given the predicate *oclIsTypeOf* parameterized with type *t* in an expression and *Class* being the variable name of the actual parsed class (invoking class), we add the conjunct *type(Class,t)* to the resulting logical sentence.

Example: (oclIsTypeOf : motherboard clockrate)

type(M, motherboard) ∧ type(M,motherboard-1) ∧
val(M,clock_rate,Clock_RateValue) ⇒
Clock_RateValue = 300).

3.12 if-then-else / let-expressions

The boolean *if-then-else* operator defined in OCL is another redundant element which can be translated to our logical notation using implications.

A *let*-expression can be employed in OCL constraints to introduce variables for subexpressions that appear more then once within the expression. We do not regard these expressions since they are merely shorthand notations, that can be replaced by the assigned subexpression.

4 IMPLEMENTATION

In order to test the applicability of our ideas we have implemented a prototype framework using standard commercial tools for this purpose. The presented ideas, however, are general enough and do not rely on some special modeling tools or inference engines to solve the configuration task.

4.1 Prototype architecture

Figure 4 shows the general architecture of the prototype implementation. We will now shortly discuss the individual components of that development environment for product configuration knowledge bases.

- **Modeling Tool (1):** As modeling tool for the conceptual product model we use *Rational Rose* which is a wide spread CASE tool for object oriented software development. Generally, any tool supporting UML can be employed since we do not add any new features to UML but rather use UML's built-in extension mechanisms. In addition, this tool can be easily customized to support the definition of domain oriented models.

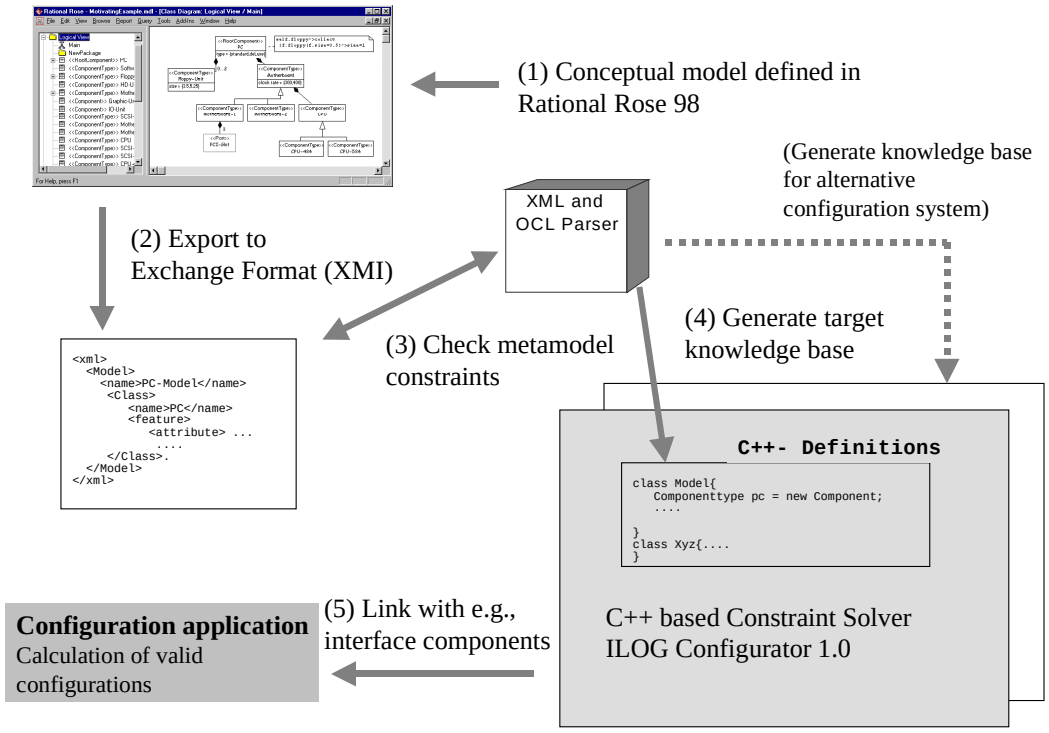


Figure 4 Prototype Implementation Architecture

- **Exchange format XMI (2):** The XML Metadata *Interchange Format (XMI,* [16]) is a standard defined by the *Object Management Group (OMG)* for a text-based exchange of UML models. The representation of the model in XMI allows us to be independent from proprietary tool formats. We transform the conceptual model from Rational Rose using a free conversion utility.
- **Checking of metamodel constraints (3):** In Section 2.1 we give examples for the definition of constraints on the correct usage of the extended modeling features for the configuration domain, e.g., a stereotyped association "*connected with*" can only be established between two stereotyped classes of type "*Port*". Since these constraints are also defined in OCL, we can analyze the model automatically and check, whether those well-formedness rules are observed. This feature allows some early check of the model defined by the designer. As a result the generated knowledge base is syntactically executable, although semantic errors are still possible. In case of semantic errors, the configurator will not return the expected results given some input, i.e., will not come up with a correct solution. Model-based diagnosis of knowledge based configurators using test examples is discussed in [6] but beyond the scope of this paper.
- **Generation of target knowledge base (4):** After the well-formedness check of the model, we can generate the target knowledge base. Therefor, we employ a XML-parser that reads the product model from the XMI-description and generates the sentences for the target knowledge base. In addition, we are developing an OCL-parser (using a standard compiler generator) that evaluates the OCL-constraints within the actual product model and transforms them according to the semantics defined in Section 3. Note that we only generate the knowledge base, where, in general, we do not have to consider the search algorithm implemented by the inference engine. Given the precise semantics of our conceptual models, knowledge bases for other configuration engines (e.g., for an integrated SAP R/3 System) can be generated.
- **Configuration engine (5):** For the actual configuration task we employ the industrial strength constraint based configuration engine *ILOG Configurator 1.0*. Since this tool consists of a set of C++ libraries, the knowledge base rather consists of C++ statements than of logical sentences. We generate the knowledge base for that software strictly conforming to the semantics defined in Section 3 and in [5]. The mapping from our logic based model to the model of the solver software is in most cases straightforward, because *ILOG Configurator* is also based on a component-port model of configuration. In principal, other configuration tools (or specialized inference engines) relying on the component-port model [7] can be used as solving components which implement other inference mecha-

nisms, e.g., functional reasoning. After the generation of the knowledge base files, these files can be embedded and linked with e.g., sme interface omponent supporting the interactive configuration task. The actual search for solutions given specific user inputs is done by the constraint solver using a backtracking algorithm with forward checking. Therefor, the knowledge base is defined in a declarative manner and separated from the problem solving task.

4.2 Experimental results

After having defined the conceptual model in Rational Rose, our implemented system generates the knowledge base definitions for the target configuration engine in a few seconds on a Pentium-II for a small sized problem of our extended working example (see [5]). After automated compilation and linkage, all solutions (a few hundred) to our simple example can be computed within less than a second, which shows the effectiveness of the generated knowledge base code.

In addition to the tests on our simplified PC-domain from our example, we have evaluated our approach on a real-world problem from the domain of private telephone switching systems. This test case showed the applicability of the conceptual modeling language as well as the effectiveness of the resulting configurator knowledge base.

When scaling to more complex products (many component types, associations, and constraints), the conceptual product models tend to get large and harder to comprehend. In these cases, the built-in structuring mechanisms of UML (and the employed CASE tools), i.e., *UML packages* and different views are employed to keep the model understandable. In cases the configuration solution process itself became harder (more variables and constraints in our representation) we did not encounter serious performance problems, because the generated code is tailored to the representation of the utilized configurator software.

Since the generated code has the form of C++ class definitions and methods, it can be easily incorporated into other applications or can be changed manually if needed.

5 RELATED WORK

The formalization of the semantics of conceptual modeling languages like OMT, UML and OCL is an actual research area ([2],[11]). These formalizations are mostly based on mathematical models and specification languages. Their work is focused on getting precise definitions of the employed concepts for general models. We view our work as complementary since our goal is to generate formal descriptions which can be interpreted by logic based problem solvers and are restricted to a special domain.

There is a long history in developing configuration tools in knowledge based systems [21] starting from early rule-based Systems (R1/XCON, [14]) to nowadays' declarative knowledge representation. 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 [12] project. In this project, specifications are developed and maintained by end-users in a declarative manner using a graphical language for the astronomical domain. The main focus of this project is to automate software reuse, where a procedural program is constructed from existing software libraries, whereas our approach uses a constraint based inference engine optimized for solving configuration problems.

CommonKADS [20] is a methodology for the development of knowledge based systems (KBS) supporting many aspects of a KBS project. Recent work incorporates actual techniques from standard software engineering processes like UML for the conceptual modeling task. Our work covers only a small part of the whole methodology (i.e., knowledge acquisition, conceptual models but no organisational tasks like project management) where the purpose of our work lies in the definition of precise semantics for a special application domain (product configuration). In addition we generate executable descriptions, where the needed inference process (knowledge) is not changing (or is given) for configuration problems.

Beside the predefined profiles for Business Modeling and for the Software Development Process, the definition of further profiles for specialized domains (e.g., scheduling, CORBA) is work in progress done by the Object Management Group (OMG). Our work differs from those standardization approaches insofar, as we define the semantics for the extensions on a rigorous formal approach.

[10] shows how UML static structure diagrams with OCL constraints can be used for the design of relational databases. The class diagrams are transformed into a relational database schema; additional OCL constraints are used to describe database integrity constraints, whereby OCL expressions are transformed to corresponding SQL-queries.

In our opinion, our approach using UML can also be utilized in other classical application fields of knowledge based systems, e.g., scheduling. Especially for the scheduling domain, an *ontology* or common vocabulary (e.g., task, resource, duration) is already established in academic and industrial communities. For the realization of our approach for scheduling, a mapping from scheduling terms to the concepts to UML has to be found; in addition the semantics of the (newly introduced) concepts have to be defined precisely. In [1], it is shown, how UML can be

employed for product data exchange. They show a mapping from an industrial product model exchange language (STEP) to concepts in UML. In our work, model exchange is done using XMI-documents, which is the emerging standard for data exchange over the Internet.

6 CONCLUSIONS

In this paper we have shown how a general modeling language from the area of Software Engineering (UML) can be employed to construct configuration knowledge bases executable by a specialized inference engine. The extensibility features from UML were employed to define a domain specific extension for the area of product configuration. For the case of complex constraints within the product structure, the UML-built-in language OCL can be employed. We showed how these expressions can be given precise semantics and can be transformed to a logical representation. Given this framework for the development of product configuration systems, the knowledge base can be defined and maintained on a conceptual level, which is comprehensible and easy-to-communicate between knowledge engineers and domain experts.

For the domain of product configuration we show how standard design techniques from industrial software development processes can be utilized for the critical and time-consuming knowledge acquisition and maintenance phases for traditional knowledge-based systems. The usage of a standardized conceptual language is made possible by the definition of precise semantics for a special domain. Consequently, the employment of adaptable knowledge based systems with clear separation of domain knowledge and inference knowledge in industrial environments is alleviated, since no special additional representation mechanisms and languages need to be employed.

7 REFERENCES

[1] F. Arnold, G. Podehl: Best of Both Worlds - A Mapping from EXPRESS-G toUML", Proc. <<UML>>'98 - Beyond the notation, Mulhouse, France, June 1998.

[2] R.H. Bourdeau, B.H.C. Cheng, A Formal Semantics for Object Model Diagrams, IEEE Transactions on Software Engineering, Vol. 21, No. 10, October 1995.

[3] R. Breu et al. : Towards a Formalization of the Unified Modeling Language, Proc. ECOOP'97, Finland, 1997.

[4] S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer Verlag Berlin Heidelberg, 1990.

[5] A. Felfernig, G. Friedrich, D. Jannach: UML as domain specific language for the construction of knowledge based configuration systems; in: Proc. SEKE'99, Kaiserslautern, 1999.

[6] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner: Consistency based diagnosis of configuration knowledgebases. *AAAI-Workshop:Configuration*, Orlando, AAAI-Press, 1999.

[7] G. Fleischanderl, G. Friedrich, A. Haselboeck, H. Schreiner and M. Stumptner, Configuring Large Systems Using Generative Constraint Satisfaction, IEEE Intelligent Systems, July/August, 1998.

[8] M. Fowler, K. Scott, UML Distilled – Applying the standard object modeling language, Addison Wesley, 1997.

[9] G. Friedrich, M. Stumptner, Consistency-Based Configuration, *AAAI'99 Workshop: Configuration*, Orlando, AAAI-Press 1999.

[10] B. Dermuth, H. Hussmann: Using UML/OCL Constraints for Relational Database Design, Proc. <<UML>>'99, Fort Collings, Colorado, 1999.

[11] A. Evans, S. Kent: Core Meta-Modeling Semantics of UML: the pUML approach, Proc. <<UML>>'99, Fort Collings, Colorado, 1999.

[12] Lowry, M., Philpot, A., Pressburger, T., Underwood, I., A Formal Approach to Domain-Oriented Software Design Environments, in Proc. 9th KBSE Conference, Monterey, CA, 1994.

[13] D. Mailharro: A Classification and Constraint-based Framework for Configuration, AI EDAM, Vol 12 (1998), Cambridge University Press, 1998.

[14] J. McDermott. R1: A Rule-Based Configurer of Computer Systems, in: Artificial Intelligence, Vol. 19, (1), p. 39-88, 1982.

[15] S. Mittal and F. Frayman, Towards a generic model of configuration tasks, Proc. IJCAI'89, pp. 1395-1401, 1989.

[16] Object Management Group (OMG), XMI Specification, http://www.omg.org, 1999.

[17] H. Peltonen, T. Männistö, K. Alho, and R. Sulonen. Product configurations—an application for prototype object approach. In Mario Tokoro and Remo Pareschi, editors, Object Oriented Programming, Proc.: ECOOP'94, pages 513–534. Springer, 1994.

[18] H. Peltonen, T. Männistö, T. Soininen, J. Tiihonen, A. Martio, and R. Sulonen, Concepts for Modeling Configurable Products. In Proc. of European Conference Product Data Technology Days 1998, p. 189-196. Quality Marketing Services, Sandhurst, UK, 1998.

[19] M. Richters, M. Gogolla: A metamodel for OCL, Proc. <<UML>>'99, Fort Collings, Colorado, 1999.

[20] G. Schreiber et al, Knowledge Engineering and Management: The CommonKADS Methodology, MIT Press, Dez. 1999.

[21] M. Stumptner, An overview of knowledge-based configuration, AI Communications 10(2), p. 111-126, 1997.

[22] J B. Warmer, A. G. Kleppe: The Object Constraint Language: Precise Modeling With UML. Addison-Wesley, 1999.