Knowledge Acquisition for Configuration Systems: UML as a link between AI and Software Engineering

Alexander Felfernig , Gerhard E. Friedrich*[x], Dietmar Jannach**

* Institute for Information Technology
University of Klagenfurt
9020 Klagenfurt, Austria

[x] Siemens Austria
Program and System Development
1030 Vienna, Austria

email: {felfernig,friedrich,jannach}@ifi.uni-klu.ac.at

ABSTRACT

In many domains, software development has to meet the challenges of developing highly adaptable software very rapidly. In order to accomplish this task, domain specific, formal description languages and knowledge-based systems are employed. From the viewpoint of the industrial software development process, it is important to integrate the construction and maintenance of these systems into standard software engineering processes. In addition, the descriptions should be comprehensible for the domain experts in order to facilitate the review process.

For the realization of product configuration systems, we show how these requirements can be met by using a standard design language (UML-Unified Modeling Language) as notation in order to simplify the construction of a logic-based description of the domain knowledge. We show how classical description concepts for expressing configuration knowledge can be introduced into UML and automatically be translated into logical sentences. These sentences are exploited by a general inference engine solving the configuration task.

KEYWORDS

Knowledge representation, automated reasoning.

1 INTRODUCTION

Shorter product cycles, lower prices of products, and higher customer demands created big challenges for 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 for clear separation between domain knowledge and inference knowledge. These techniques can be exploited to (partially) automate the generation of software solutions.

Unfortunately, in many cases, such high level, formal description languages are not integrated in the industrial software development process. In addition, these descriptions are difficult to communicate to domain experts for reviewing purposes. This makes it demanding for software development departments to incorporate such technologies into their standard development process.

Therefore, our goal is to make such descriptions more accessible both for the software engineering practitioners and domain experts with a technical background.

As an application domain we introduce product configuration systems for two important reasons. First, the industrial demand for such systems is high and increasing, e.g., in the telecommunication and computer industry. Second, product configuration creates big challenges on software development.

- The complexity of the task requires the sophisticated knowledge of technical experts.
- The configuration has to be adapted continuously because of changing components and configuration constraints.
- Configurator development time and maintenance time are short and strictly limited. Development of the product and the product configuration system has to be done concurrently.

In order to enhance the usability of formal descriptions we employ UML [5] for two reasons. First, OMT and UML (as its successor) are widely applied in industrial software development as a standard design method. Second, we made excellent experiences in using OMT-designs for validation by technical domain experts.

The key idea of our approach is twofold: First, we extend the static model of UML by broadly used configuration-specific modeling concepts. Second, we define a mapping from these concepts to a configuration language based on first-order-logic. Consequently, the construction of a logic-based description of the domain knowledge is simplified.

We employ the extension mechanism of UML (*stereotypes*) to express domain-specific modeling concepts, which has shown to be a promising approach in other areas [14]. The semantics of the different modeling concepts are formally defined by the mapping of the notation to logical sentences.

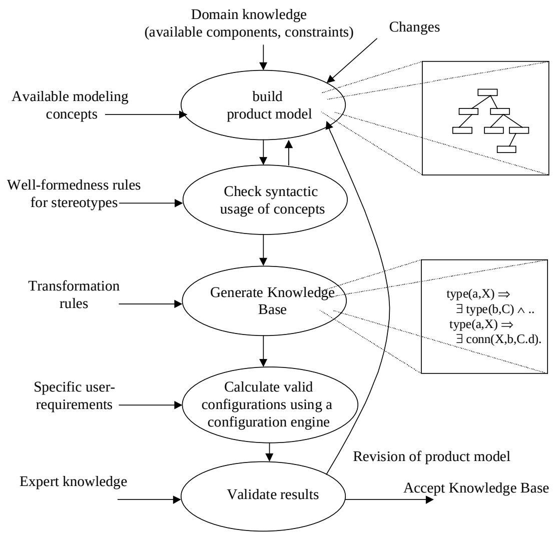
Domain knowledge
(available components, constraints)
Changes
Available modeling concepts
build product model
Well-formedness rules for stereotypes
Check syntactic usage of concepts
Transformation rules
Generate Knowledge Base
type(a,X) ⇒ ∃ type(b,C) ∧ ..
type(a,X) ⇒ ∃ conn(X,b,C.d).
Specific user-requirements
Calculate valid configurations using a configuration engine
Revision of product model
Expert knowledge
Validate results
Accept Knowledge Base

Figure 1 Configurator development process

The development process for these configuration systems is shown in Figure 1. The UML-model is non-ambiguously transformed to logical sentences which are exploited by a general configuration engine [4] for computing configurations of products. Consequently, the configurator is based on a declarative, logic based, explicit representation of the configuration knowledge.

The contributions of this paper are broadening the scope of formal approaches as well as the applicability of logic-based, declarative technology within the standard software development process. In addition, we facilitate rapid application development, and enhancement of the validation and maintenance tasks because these tasks are performed on a conceptual model.

The rest of the paper is organized as follows. After giving a motivating example (chapter 2), we describe the underlying logical model of a configuration problem (chapter 3). In chapter 4 we describe typical modeling concepts for product configuration, their representation in UML and the transformation to logical sentences. Chapter 5 describes the application environment. Finally, chapter 6 and 7 contain related work and conclusions.

2 MOTIVATING EXAMPLE

The following example shows how a configurable product can be modeled using an UML static structure diagram. This 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.

For presentation purposes we introduce a simplified model of a configurable PC 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 [12]. Additionally, positive application tests were conducted in the telecommunication domain.

Note, that compared to the OO-analysis in unstructured domains, classes (component-types) in the configuration domain are easily identified.

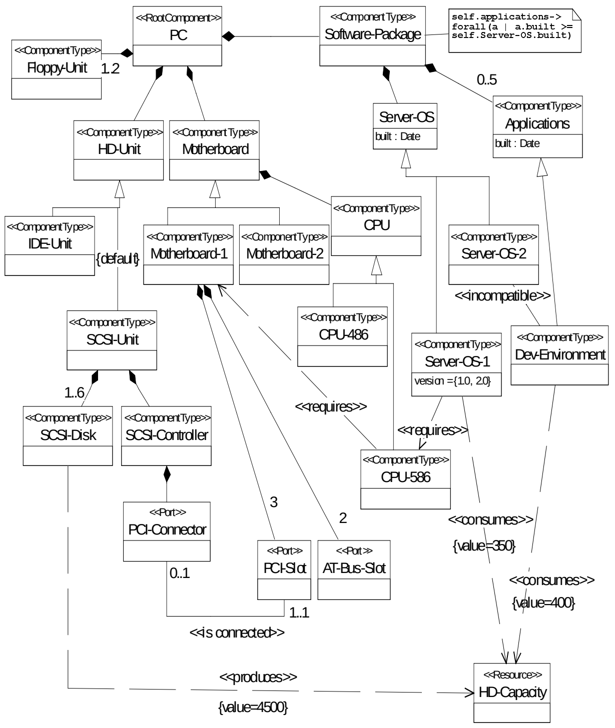
<<RootComponent>> PC
<<ComponentType>> Software-Package
self.applications->forall(a | a.built >= self.Server-OS.built)
<<ComponentType>> Floppy-Unit
1..2
0..5
Server-OS
built : Date
<<ComponentType>> Applications
built : Date
<<ComponentType>> HD-Unit
<<ComponentType>> Motherboard
<<ComponentType>> CPU
<<ComponentType>> IDE-Unit
{default}
<<ComponentType>> Motherboard-1
<<ComponentType>> Motherboard-2
<<ComponentType>> Server-OS-2
<<incompatible>>
<<ComponentType>> CPU-486
<<ComponentType>> SCSI-Unit
<<ComponentType>> Server-OS-1
version ={1.0, 2.0}
<<ComponentType>> Dev-Environment
1..6
<<ComponentType>> SCSI-Disk
<<ComponentType>> SCSI-Controller
<<requires>>
<<requires>>
<<ComponentType>> CPU-586
3
2
<<Port>> PCI-Connector
<<Port>> PCI-Slot
<<Port>> AT-Bus-Slot
<<consumes>>
{value=350}
<<consumes>>
{value=400}
0..1
1..1
<<is connected>>
<<produces>>
{value=4500}
<<Resource>> HD-Capacity

Figure 2 Product model of a configurable PC

Additional modeling concepts for configuration

Requires and incompatible The set of valid PC-configurations is restricted through stereotyped requires-relations and incompatible-relations between different components.

Ports and connections For some configuration domains, not only the quantity and kind of the employed components are important but also how different components are connected to each other.

In our example, an SCSI-controller has a port (expressed through a stereotype class) called "PCI-connector". A motherboard of type "Motherboard-1" has three PCI-slots. The multiplicity of the stereotyped association "is connected" denotes, that a PCI-connector must be connected to a PCI-slot, whereas a PCI-slot can possibly be connected to a PCI-connector.

Resources A further enhancement of the model is expressed through resources which impose additional constraints on the possible product structure.

The contribution and consumption of a resource is modeled through relations "consumes" and "produces". A tagged value denotes the actual value of production and consumption. In our example, the disk-capacity of the system must be greater or equal to the capacity consumed by the installed software.

Calculating configurations

After having defined the configurable product, the actual configuration can take place. The user (the customer) can provide some input data and specify the requirements for the actual variant of the product.

Let the customer requirement be that the development environment has to run on the system, i.e., that this component has to be part of the final product. Starting from this input, a configuration system builds a valid solution (Figure 3) meeting these requirements.

The next chapter shows a formal definition of a configuration problem that serves as a basis for many existing configuration systems. In chapter 4, we show how the generic model can be transformed to a logic theory for a configuration system built upon these definitions.

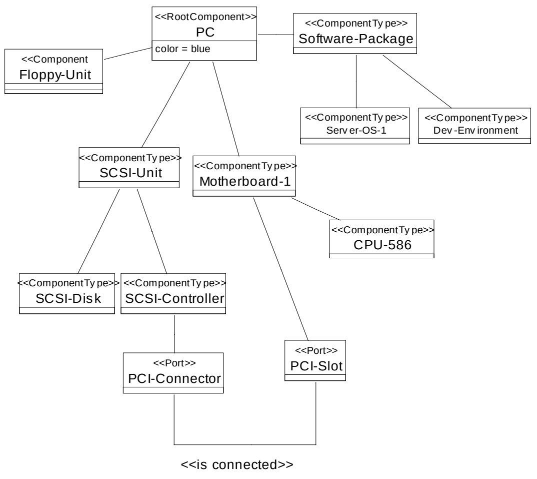

Figure 3 A configuration as an instance model

3 CONFIGURATION PROBLEM

The following 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. Components are described by attributes and ports. Ports are used as connection points between components [11].

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 ([11][15][16]). The model will now be treated more formally.

The formulation of a **configuration problem** can be based on two sets of logic sentences, namely DD (domain description) and SRS (specific requirements). 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. Additionally, domain-specific axioms for configuration are defined, e.g., one port can only be connected to exactly one other port.

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

An example from the PC-configuration:

types = {pc,cpu,motherboard,...}.
attributes(server-os-1) = {version}.
dom(server-os-1,version)= {1.0,2.0}.
ports(pc) = {hd-unit-port, motherboard-port,... }.
ports(motherboard) = {pc-port,cpu-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.

1. **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.

2. **CONNS** is a set of literals of the form *conn(c1,p2,c2,p2)*. *c1* and *c2* are component identifications from COMPS, *p1* (*p2*) is a port of the component *c1* (*c2*).

3. **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(p1,pc).
type (m1,motherboard-1).
type(c1,cpu-586).
conn(p1,motherboard-port,m1,pc-port).
conn(c1,motherboard-port,m1,cpu-port).

Note, that component *p1* of type *pc* has a port named "motherboard-port" reserved for connections to a motherboard. This port is defined in the domain description.

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 $\cup$ SRS $\cup$ COMPS $\cup$ CONNS $\cup$ 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 the domain description. 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 [7].

4 TRANSFORMATION RULES

In order to allow automatic construction of the knowledge base from the conceptual model, we have to clearly define the semantics of the employed concepts. In our approach, we define the semantics through logical sentences for the configuration model defined in chapter 3.

These logical sentences[1] restrict the set of possible configurations, i.e., instance models which strictly correspond to the class diagram defining the product structure. The result of the transformation is a set of first-order logical sentences that form a domain description that can be used by a configuration system.

Component-types

Component-types describe the predefined parts a product is built of. We use a stereotype class for representing components since some limitations on these classes have to hold (e.g., there are no methods, attributes are limited to simple data types and enumerations). For each component-type in the UML-model, we extend the domain description as follows.

Definition: Given a component-type *c* in the graphical representation (GREP) then $c \in$ *types.*

Given an attribute *a* of component-type *c* in GREP then $a \in$ *attributes(c).*

Given a domain description *d* of an attribute *a* of component-type *c* in GREP then *dom(c,a) = d*.

Generalization

Subtyping in the configuration domain means, that attributes, ports and constraints are inherited to the subtype. We assume disjunctive semantics for generalization which is also the default semantics in UML, i.e., in a configuration only one of the given subtypes will be instantiated. Additionally, no multiple inheritance is allowed in order to facilitate comprehensible semantics.

Definition: Let *u* and $d_1,...,d_n$ be classes where *u* is the superclass of $d_1,...,d_n$ and *c* is the set of all direct and indirect superclasses of *u* in GREP then
for i = 1,...,n

- the domain description is extended as follows:

  $type(ID,d_i) \Rightarrow type(ID,u).$
  $type(ID,u) \Rightarrow type(ID, d_1) \vee,...,\vee type(ID, d_n).$
  $type(ID,X) \wedge type(ID,Y) \wedge X \in \{ d_1,...,d_n \} \Rightarrow$
  $\quad Y \in (\{ u\} \cup c) \vee X=Y.$

- If $a \in attributes(u)$ then $a \in attributes(d_i)$.
- If $p \in ports(u)$ then $p \in ports(d_i)$.

Example for extensions to the knowledge base:

type(ID,cpu-486) ⇒ type(ID,cpu).
type(ID,cpu) ⇒
type(ID,cpu-486) ∨ type(ID,cpu-586).

Part-Refinement

UML differentiates between shared and composite aggregation. The semantic difference between aggregation and composition in the UML-Semantics-Definition gives some room for interpretation.

In the case of configuration modeling, semantics can be defined as follows: If a component is a compositional part of another component, we require *strong ownership* and it can not be part of another component at the same time. If a component is a non-composite part of another component, we say that this component can be *shared* among different other components.

The multiplicity of the aggregation denotes of how many parts the aggregate can consist of and between how many aggregates a part can be shared if the aggregation is non-composite.

[1] 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. We use the unique name assumption except for skolem constants.

The aggregation relationship is modeled in the component-port-model through introduction of ports for connecting the aggregate with its parts (see Figure 4).

For the following definitions no cycles in the part-of structure are allowed. In the component-port-model connections may only be established between ports and each port can be connected to exactly one other port.

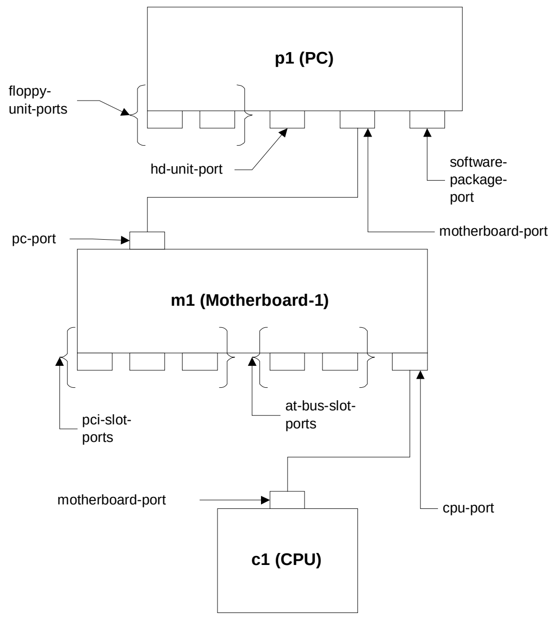

Figure 4 Aggregation in the component port model

Composite aggregation

First, we extend the port definitions of the affected component-types. Ports are defined for the aggregate in the amount of the upper bound of the multiplicity of the part. Only one port is added to the part component-type to connect it with the aggregate. The ports are named according to the name of the aggregation. If no association name is specified, the name of the opposite component-type is used. The name can denote different roles a part can play in the aggregate.

Second, we derive logical sentences stating that, if an aggregate is in the configuration, components in the amount of at least the lower bound of the multiplicity of the part must be added too. Each part must be connected to (be part of) exactly one aggregate if the multiplicity of the aggregate is "1..1". If the multiplicity is "0..1", which is the only other possibility defined in UML, then no connection has to be established.

Definition: Let *w* and *p* be two component-types in GREP where *p* is a compositional part of *w* and *ub* is the upper bound and *lb* is the lower bound of the multiplicity of the part. Let *name* be the name of the association. We have to extend our configuration description in a way that:

$\{name_1,..., name_{ub}\} \subseteq ports(w).$
$name \in ports(p).$

Example:

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

At least *lb* parts must exist and be connected and the following constraint is derived:

$type(ID,w) \Rightarrow \bigwedge_{i=1,...,lb} ((\exists ID_i, Port\_part_i)\ type(ID_i,p) \wedge conn(ID, Port\_part_i, ID_i, name) \wedge Port\_part_i \in \{name_1,..., name_{ub}\}) \wedge ID_1 \neq ,..., \neq ID_{lb}.$

The parts have to be connected with the aggregate:

$type(ID\_part,p) \Rightarrow \exists (ID\_agg, Port\_part)\ type(ID\_agg,w) \wedge Port\_part \in \{name_1,..., name_{ub}\} \wedge conn(ID\_part, name, ID\_agg, Port\_part).$

Example:

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

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

Shared aggregation

In the case of shared aggregation, additional ports have to be defined for the part, because the part can be part of (connected to) more than one aggregate. Connections have to be established according to the lower bound of the multiplicity.

Definition: Let *w* and *p* be two component-types in GREP where *p* is an aggregate part of *w* and *ubpart* is the upper bound and *lbpart* is the lower bound of the multiplicity of the part and *ubagg* is the upper bound and *lbagg* is the lower bound of the multiplicity of the aggregate. Let *name* be the name of the associations. We have to extend our configuration description in a way that:

$\{name_1,..., name_{ubpart}\} \subseteq ports(w).$
$\{name_1,..., name_{ubagg}\} \subseteq ports(p).$

We denote the set $\{name_1 ,..., name_{ubagg}\}$ as *ports(p,name).*

A constraint is derived stating that at least *lbpart* ports have to be connected with different parts:

$type(ID,w) \Rightarrow \bigwedge_{i=1,...,lbpart} ((\exists ID_i, Port\_part_i, Port\_agg_i)\ type(ID_i,p) \wedge conn(ID, Port\_part_i, ID_i, Port\_agg_i) \wedge Port\_part_i \in \{name_1,..., name_{ubpart}\} \wedge Port\_agg_i \in \{name_1,..., name_{ubagg}\}) \wedge ID_1 \neq ,..., \neq ID_{lbpart}.$

At least *lbagg* ports of the part have to be connected with the aggregate. If the lower bound is zero, then no connections are established:

$type(ID,p) \Rightarrow \bigwedge_{i=1,...,lbagg} ((\exists ID_i, Port\_agg_i, Port\_part_i)\ type(ID_i,w) \wedge conn(ID, Port\_agg_i, ID_i, Port\_part_i) \wedge Port\_agg_i \in \{name_1,..., name_{ubagg}\} \wedge Port\_part_i \in \{name_1,..., name_{ubpart}\}) \wedge ID_1 \neq ,..., \neq ID_{lbagg}.$

The following additional constraints have to be added to the domain description to define the semantics of aggregation clearly. First, if a component is a compositional part of an aggregate, it can not be a part of any other component at the same time. Second, an instance of a component-type being part of any part-of relationship must be connected to an instance of an aggregate type.

Definition: Let p, $a_1,...,a_n$, $c_1,...,c_m$ be component-types where p is an aggregational part of $a_1,...,a_n$ and a compositional part of $c_1,...,c_m$ in GREP. Let $name\_agg_1,...,name\_agg_n$ be the names of the aggregate associations in GREP, and $name\_comp_1,...,name\_comp_m$ be the names of the composition associations.

Let *all_part_of_ports* be $\{name\_comp_1,...,name\_comp_m\} \cup ports(p,name\_agg_1) \cup ,..., \cup\ ports(p,name\_agg_n)$.

To forbid other connections if one composite port is connected, the following constraint has to hold:

$$\mathbf{type(ID,p) \wedge conn(ID,C,\_,\_) \wedge C \in \{name\_comp_1,...,name\_comp_m\} \wedge D \in all\_part\_of\_ports \wedge conn(ID,D,\_,\_) \Rightarrow C = D.}$$

At least one of the ports must be connected.

$$\mathbf{type(ID,p) \Rightarrow \exists (Port, X)\ Port \in all\_part\_of\_ports \wedge conn(ID,Port,X,\_) \wedge type(X,W) \wedge W \in \{a_1,...,a_n, c_1,...,c_m\}.}$$

Presuppositions on the part-of hierarchy

Any of the following constraints on the product structure derived from GREP must ensure that the involved components are within the same sub-configuration w.r.t. the part-of hierarchy, i.e., the involved components must be connected to the same instance of the component-type that represents the common root for these components. For a correct derivation of constraints, we postulate that the involved component types have a unique common component-type as predecessor and a unique path to this common root in GREP. All part-of relations within the common subtree must be compositions in order to ensure uniqueness of the common predecessor on the instance level. If this property is not satisfied, the meaning of the modeling concepts is ambiguous since a part can be part of different sub-structures in the part-of hierarchy. To eliminate this ambiguity, additional modeling concepts can be defined, allowing the domain expert to express further problem specific constraints.

For the derivation of constraints, we use the abbreviations (similar to macros) *navigation_expr* and *generating_expr*, which represent a path-expression through conn-predicates from a component to an instance of the common root. In the case of *generating_expr* the variables are existentially quantified and the expression may only be used on the right-hand-side of the implications.

For the definition of these two abbreviations, we view the class-model as a directed graph, where the component-types are the vertices V and the part-of relations are the edges E. We employ the graph using the inheritance property of ports, i.e., the inheritance of part-of relations, e.g., the component-type "CPU-586" has a port inherited from "CPU" to connect it with the motherboard. Because of this property, the part-of relations are inherited to the leave nodes of the generalization hierarchy. Therefore, the generalization hierarchy does not need to be considered for the construction of the path expression.

Let *path(a,p)* describe the path from component-type *a* to the common root *p* in GREP through an ordered list of predicates of the form *part_of(component-type-a, component-type-b, association-name).*

The following formula shows how *navigation_expr* and *generating_expr* are defined.

Given $\mathbf{path(a,p) = \langle part\_of(a,y,name\_y),...,part\_of(z,p,name\_p)\rangle}$
in GREP then *navigation_expr(ID_a, P)* is defined as

$$\mathbf{conn(ID\_a,name\_y,ID\_y,\_) \wedge ,..., \wedge conn(ID\_z,name\_p,P,\_).}$$

and *generating_expr(ID_a, P)* is defined as

$$\mathbf{\exists (ID\_y ,..., ID\_z)\ conn(ID\_a,name\_y,ID\_y,\_) \wedge ,..., \wedge conn(ID\_z,name\_p,P,\_).}$$

P is a variable identifying an instance of the type of the common root.

Example:

If $\mathbf{path(cpu\text{-}586,pc) = \langle part\_of(cpu\text{-}586, motherboard,\_), part\_of(motherboard, pc,\_)\rangle}$
is the path from a CPU-586 to the PC in GREP then *navigation_expr(ID_CPU,P)* is

$$\mathbf{conn(ID\_cpu,motherboard,ID\_motherboard,\_) \wedge conn(ID\_motherboard,pc,P,\_).}$$

Requires

A relation *a requires b* in GREP denotes that the existence of an instance of component-type *a* requires that an instance of *b* exists and is part of (connected to) the same (sub-)configuration.

In our example, the fact that Server-OS-1 requires a CPU-586 implies also that Server-OS-1 and CPU-586 are part of the same PC which is the common and unique root of both component-types in the part-of hierarchy.

Definition: Given the relation *a requires b* where *a* and *b* are component-types in GREP, we extend our domain description with the following formula:

$$\mathbf{type(ID\_a,a) \wedge navigation\_expr(ID\_a, P) \Rightarrow \exists (ID\_b)\ type(ID\_b,b) \wedge generating\_expr(ID\_b, P).}$$

Example:

$$\mathbf{type(ID,server\text{-}os\text{-}1) \wedge conn(ID,software\text{-}package, S, \_) \wedge conn(S,pc,P,\_) \Rightarrow \exists(C)\ type(C,cpu\text{-}586) \wedge \exists (M)\ conn(C,motherboard,M,\_) \wedge conn(M,pc,P,\_).}$$

The left-hand side of the implication describes a path to the common root (PC). The right-hand side of the implication requires the existence and connection of the components on the path from *b* to the common root.

We defined the semantics of a "requires" relation to be "not exclusive" and of multiplicity "1..1". If more than one component requires a component of type *b*, only one instance of b is needed. Variations of the semantics of a "requires" relation can be introduced, e.g., introduction of a multiplicity expressing that several instances of component are required.

Incompatible
This relation denotes the fact that two components can not be used within the same configuration. The incompatible relation is defined as a binary relation with a multiplicity of "1..1" in the UML-model.

Definition: Given the relation *a incompatible_with b* in GREP where *a* and *b* are component-types we extend the domain definition with the following constraint:

type(ID_a,a) ∧ navigation_expr(ID_a, P) ∧ type(ID_b,b) ∧ navigation_expr(ID_b, P) ⇒ false.

Note: If there exists a path through connections from components ID_a and ID_b to the common root (P), then *false* is derived.

Example:
type(ID,dev-environment) ∧ conn(ID,software-package, S, _) ∧ type(OS,server-os-2) ∧ conn(OS,software-package,S,_) ⇒ false.

Ports and Connections
Ports in the UML-model represent physical connection-points between components. These ports are added to the port definitions of the components. Possible and required connections are expressed through the stereotyped relation "is connected".

Definition: Let *c* be a component-type and *p* be a port where p is a part of *c* in GREP and where *n* is the multiplicity of the port. We extend **DD** as follows:

$\{p_1, ..., p_n\} \subseteq$ ***ports(c)***

We denote the set $\{p_1, ..., p_n\}$ by *ports(c,p).*

Definition: Let *a* and *b* be component-types and *pa* and *pb* be ports, where *pa* is a port of *a* and *pb* is a port of *b* and *pa* and *pb* are connected in GREP.

If the multiplicity of *pb* is “1..1”, expressing that the port must be connected, the following constraint is derived:

type(ID_a,a) ∧ navigation_expr(ID_a, P) ∧ Port_a ∈ ports(a,pa) ⇒
∃ (ID_b, Port_b) type(ID_b,b) ∧ generating_expr(ID_b, P) ∧ Port_b ∈ ports(b,pb) ∧ conn(ID_a, Port_a, ID_b, Port_b).

The definition is much the same as for "requires", because if we want to connect a port *Port_a* from *a* with a port *Port_b* of component *b*, the existence of an instance of type *b* is required. Only the additional connection has to be established.

If the multiplicity of *pb* is “0..1”, the following constraint is derived:

type(ID_b,b) ∧ navigation_expr(ID_b, P) ∧ Port_b ∈ ports(b,pb) ∧ conn(ID_b, Port_b,ID_a,Port_a)⇒
∃ (ID_a, Port_a) type(ID_a,a) ∧ generating_expr(ID_a, P) ∧ Port_a ∈ ports(b,pa).

In this sentence we define that, if a component of type *b* exists and a connection from *Port_b* is established, then this connection must be to a port of a component of type *a*.

Example for the PCI-Connector:

type(S,scsi-controller) ∧ conn(S,scsi-unit,U,_) ∧ conn(U,pc,P,_) ∧ Port_S ∈ ports(scsi-controller,pci-connector) ⇒
∃ (M,Port_M) type(M,motherboard-1) ∧ Port_M ∈ ports(motherboard-1,pci-slot) ∧ conn(S,Port_S,M,Port_M).

Resources
Resource constraints are modeled in the UML-model through stereotyped classes representing types of resources and stereotyped relations indicating production and consumption of these resources. Resources represent a balancing task [9] within the shared subtree of the part-of hierarchy of the product structure.

To map the resource task to the component-port model, additional attributes have to be defined for the participating component-types holding the actual value of production and consumption. A constraint has to be derived that ensures, that resource values consumed and produced by components are balanced. In the example given, the component-type PC is the unique common root for all consumers and producers of the resource hard disk capacity. A constraint for instances of a PC is constructed, ensuring that the sum of the produced capacity exceeds the sum of the consumed capacity. We therefore collect all the instances of SCSI-Disks that are part of this PC and the consumers that are part of the PC using the predicates

allconsumers(result_set,ID_Root) and

allproducers(result_set,ID_Root).

These predicates return a set of instances of consuming and producing components connected to the actual instance of the root component.

Definition: Let $g_1, ..., g_n$ be producing component-types of resource *r* with attribute values gv_i and $c_1, ..., c_m$ be consuming component-types with values cv_i. The values of cv_i and gv_i are determined by the tagged values of the "consumes" and "produces" relations.

We have to extend the domain description as follows:

$r \in \mathit{attributes}(g_i)$, for i = 1 to n.
$r \in \mathit{attributes}(c_i)$, for i = 1 to m.
$\mathit{val}(g_i, r, gv_i)$ for i = 1 to n.
$\mathit{val}(c_i, r, cv_i)$ for i = 1 to m.

Let *p* be the common and unique predecessor w.r.t. the part-of-hierarchy of all consumers and producers. We derive the following constraint:

$$\mathit{type}(P,p) \wedge \mathit{allconsumers}(Consumer,P) \wedge \mathit{allproducers}(Producer,P) \Rightarrow \sum_{(o \in Consumer \wedge val(o,r,V))} V <= \sum_{(s \in Producer \wedge val(s,r,W))} W.$$

The predicates *allconsumers* and *allproducers* are defined as follows using LDL-notation [3]:

$$\begin{aligned}&\mathit{allconsumers}(<Consumer>,P) \Leftarrow\\ &C \in \{c_1,...,c_n\} \wedge \mathit{type}(Consumer,C) \wedge\\ &\quad \mathit{navigtion\_expr}(Consumer,P) \wedge \mathit{type}(P,p).\end{aligned}$$

$$\begin{aligned}&\mathit{allproducers}(<Producer>,P) \Leftarrow\\ &G \in \{g1,...,gn\} \wedge \mathit{type}(Producer,G) \wedge\\ &\quad \mathit{navigation\_expr}(Producer,P) \wedge \mathit{type}(P,p).\end{aligned}$$

Note: All component instances with one of the correct types are collected within *<Consumer>*. *navigation_expr(Consumer,P)* ensures that all these components are connected to the same instance *P* of the common root.

Additional modeling concepts and constraints

Chapter 2 gives an overview of modeling concepts used in modeling of configuration domains. These concepts have shown to cover a wide range of application areas for configuration [13].

Despite this, some application areas may have a need for special modeling concepts not covered so far. To introduce a new modeling concept the following two steps have to be taken: First, define the new concept (a new stereotype) and state the well-formedness rules for its correct use within the model. Second, define the semantics of the concept for the configuration domain by stating the facts and constraints induced to the logic theory when using the concept.

If there are additional constraints in the configurable product, which can not be expressed through predefined or newly introduced graphical concepts, the knowledge base has to be extended by a knowledge engineer.

When defining these transformation rules or adding other constraints to the knowledge base, one has to consider the structure of the derived constraints, which must conform to the restrictions mentioned in chapter 3 to allow for decidability and executability.

Example:

For the constraint "the built-date of applications must be greater than the built-date of the server-os", we have to add the following logical sentence:

$$\mathit{type}(SP,sw\text{-}package) \wedge \mathit{type}(S,server\text{-}os) \wedge \mathit{type}(A,applications) \wedge \mathit{conn}(SP,server\text{-}os,S,sw\text{-}package) \wedge \mathit{conn}(SP,AP,A,sw\text{-}package) \wedge AP \in \{applications\text{-}1,...,applications\text{-}5\} \wedge \mathit{val}(A,built,AB) \wedge \mathit{val}(S,built,SB) \wedge AB <= SB \Rightarrow \mathit{false}.$$

5 CONFIGURATION TOOL: COCOS

The notion of the component-port-model is well-established for modeling and solving configuration problems [11]. In general, consistency-based tools basing on the component-port-model can use the logic theory derived from the UML-product-model. Since the output is a set of logical sentences, it can be transformed to the representation of different tools doing the actual configuration task.

In fact, the configuration tool COCOS [16] relies on the component-port-model and has proven its usefulness in the configuration of large-scale electronic systems. By using a high level description language for expressing the configuration constraints we were able to reduce development effort (by 66% compared to a previous project) and maintenance costs significantly. In addition, the flexibility of the software tool with respect to human-computer interactions is enhanced (details are described in [4]). COCOS is a general configuration engine that uses a knowledge base consisting of logical sentences shown in section 4 as input and builds possible configurations.

In addition to these logical constructs, COCOS allows the definition of orderings, defaults, and preferences for instantiation of components and generation of connections. This control knowledge is specified by priority values, which can be easily incorporated in our UML-notation by additional attributes.

6 RELATED WORK

Bourdeau and Chen [1] give a formal semantics for object model diagrams based on OMT. This work is an important step in automating the process of obtaining a formal description from the information in the diagrams. They use the Larch Shared Language as their target language since their goal is to support the assessment of requirement specifications in general. A step towards the formalization of UML basing on a mathematical system model is done in [2]. We view our work as complementary since our goal is to generate formal descriptions which can be interpreted by logic based problem solvers.

Peltonen, Männistö, Alho, and Sulonen ([12], [13]) use product configuration as a practical application for a prototype based approach. They view configuration as a process where objects are created by specifying their parent and the inheritance of information. Our approach is in the tradition of explicitly describing a valid configuration using a declarative language. The process of generating configurations is an automated search process which can be guided by heuristics. Using this approach we have a clear separation of procedural and declarative knowledge as well as a precise semantics of the configuration problem and the content of the knowledge base.

There is a long history in developing configuration tools in knowledge based systems (see [15]). 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 project [9]. In this project, specifications are developed and maintained by end-users in a declarative manner using a graphical language for the astronomical domain. The specification is tested for solvability and program synthesis is done by a theorem prover, which deduces an intermediate program, that is translated to subroutine calls of a target language. 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.

Consistency management for complex applications as discussed by Tarr and Clarke [17] and product configuration share some interesting common properties. Both application areas have to ensure that an object model is consistent with respect to a set of constraints. However, in consistency management, the goal is mainly to cope with a steady stream of changes resulting in consistency violations. The repair of these violations is a main task. In product configuration, the main task is to generate a valid configuration given some customer requirements. Since our formalism is based on rigorous logic definition, concepts for representing repair actions in order to restore consistency from other consistency based models are applicable. General concepts were developed in the field of model-based diagnosis [6].

Structural information (components and ports) plays also an important role in the domain of Architecture Description Languages (ADL). Some important work was done in [8], where common concepts for ADLs and an interchange format are defined. Architectures are (graphically) modeled and represented in first-order-logic allowing for the definition of assertions and additional constraints. The architecture description may then be translated to other existing ADLs. The full semantics of the modeling concepts may vary, depending on the translation to other ADLs. However, in our approach, we use logic not only for representation and model exchange, but have the possibility to interpret the logical sentences derived from the conceptual model.

7 CONCLUSIONS

Extensible standard design methods (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 methods like the UML static model 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. This 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 mastered by software engineering of these products. In addition, our concepts correspond to the generation and consistency management of object networks which is related to various other domains.

By using UML as a front-end for formal descriptions, both sides broaden their application area - UML as a notation for formal problem specification and formal descriptions in standard industrial software development processes.

ACKNOWLEDGEMENTS

We thank Roland Mittermeir for valuable remarks and discussions.

REFERENCES

[1] 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.

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

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

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

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

[6] G. Friedrich, G. Gottlob, and W. Nejdl, Formalizing the repair process - extended report, in: Annals of Mathematics and Artificial Intelligence, Vol 11 (1994), pp.187-202, J.G. Baltzer AG, 1994.

[7] G. Friedrich, M. Stumptner, Consistency-Based Configuration, University Klagenfurt, Technical Report KLU-IFI-98-5, 1998.

[8] D. Garlan, R. T. Monroe and D. Wile, Acme: An Architecture Description Interchange Language, Proc. CASCON'97, p. 169-183, Canada, 1997.

[9] M. Heinrich and E.W. Jüngst, A resource-based paradigm for the configuration of technical systems from modular components, in Proc. 7th IEEE Conference on AI Applications (CAIA), p. 257-264, 1991.

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

[11] S. Mittal and F. Frayman, Towards a generic model of configuration tasks, Proc. Eleventh Int. Joint Conf. Artificial Intelligence, pp. 1395-1401, 1989.

[12] 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, 8th European Conference, ECOOP'94, pages 513–534. Springer-Verlag, 1994.

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

[14] J.E. Robbins, N. Medvidovic., D.F. Redmiles, D.S. Rosenblum, Integrating Architecture Description Languages with a Standard Design Method, Proc. 20th Intl. Conf. on Software Engineering, Kyoto, Japan, 1998.

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

[16] M. Stumptner, A. Haselböck, and G. Friedrich, Cocos: A Tool for Constraint-Based, Dynamic Configuration, Proc. CAIA '94: 10th Conf. AI for Applications, IEEE Computer Society Press, Calif. 1994, pp. 373-380.

[17] P. Tarr, L. Clarke, Consistency Management for Complex Applications, Proc. 20th Intl. Conf. on Software Engineering, Kyoto, Japan 1998.