Conceptual modeling for configuration of mass-customizable products

Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach[1]

Universität Klagenfurt

Institut für Wirtschaftsinformatik und Anwendungssysteme

Computer Science and Manufacturing Research Group

Universitätsstraße 65, A-9020 Klagenfurt, Austria

Email: dietmar@ifit.uni-klu.ac.at

Tel: +43 (0) 463 2700 3757

Fax: +43 (0) 463 2700 3799

Abstract. The development and maintenance of product configuration systems is faced with increasing challenges caused by the growing complexity of the underlying knowledge bases. Effective knowledge acquisition is needed since the product and the corresponding configuration system have to be developed in parallel. In this paper we show how to employ a standard design language (Unified Modeling Language - UML) for modeling configuration knowledge bases. The two constituent parts of the configuration model are the component model and a set of corresponding functional architectures defining which requirements can be imposed on the product. The conceptual configuration model is automatically translated into an executable logic representation. Using this representation we show how to employ model-based diagnosis techniques for debugging faulty configuration knowledge bases, detecting infeasible requirements, and for reconfiguring old configurations.

Keywords. Product Configuration, Conceptual Modeling, Knowledge Acquisition, Diagnosis.

Introduction

Product configuration systems play an important role in the support of the mass customization paradigm [20]. The increasing complexity and size of configuration knowledge bases requires the provision of advanced methods supporting the configurator development process as well as the actual configuration process. Informally, configuration can be seen as a design activity, where the configured product is built from a predefined set of

[1] Contact author

component types that can be parameterized and interconnected on pre-enumerated connection points. Additional constraints are used to restrict the number of legal product constellations.

There exists a variety of application areas for product configuration systems, e.g., in the computer industry (PC configuration), telecommunication industry (configuration of switching systems), or automotive industry (car sales configuration).

Figure 1 shows the three main components of our proposed configuration environment. Knowledge acquisition is done using configuration domain specific modeling concepts represented as UML stereotypes [8]. UML [24] is a conceptual modeling language, which is widely applied in industrial software development processes. This notation is similar to OMT diagrams (Object Modeling Technique) [23] and is easy to understand and communicate to domain experts. The resulting models are automatically translated into a logical representation executable by a configuration engine. After having designed and translated the configuration model (knowledge acquisition), the resulting configuration knowledge base has to be validated. We support this task in our framework using model-based diagnosis techniques: Given positive and negative configuration examples (that can in turn be modeled on the conceptual level and transformed to a logical representation) we identify parts of the resulting knowledge base causing an unexpected behavior of the configuration system. The outcome of this validation phase is a set of logical sentences from the generated knowledge base that have to be revised in order to correct the knowledge base. Note, that these results can be easily related to the pieces of knowledge from the (graphical) conceptual model. Therefore, the adaptation of the configuration knowledge can be done again on the conceptual level.

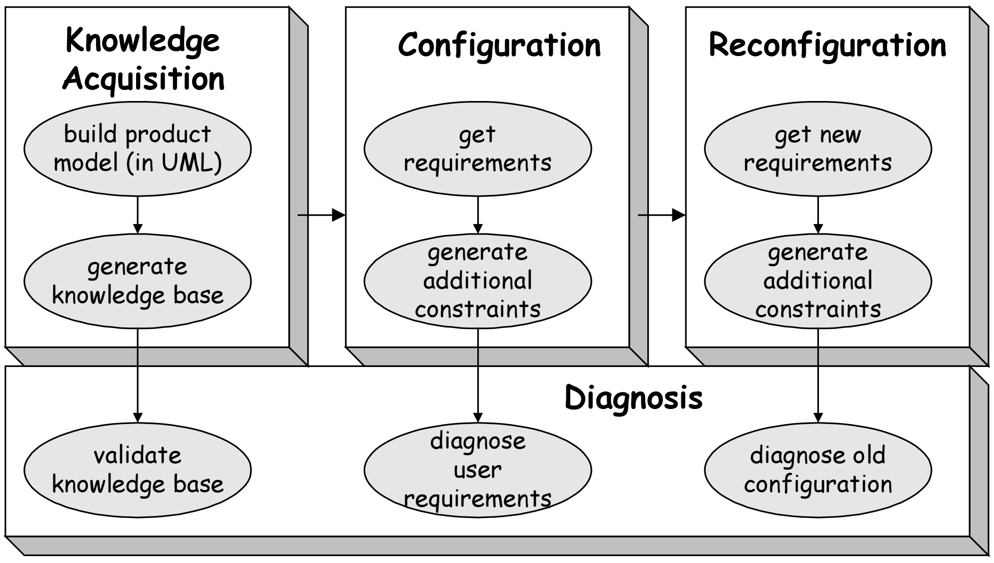

Knowledge Acquisition
build product model (in UML)
generate knowledge base
Configuration
get requirements
generate additional constraints
Reconfiguration
get new requirements
generate additional constraints
Diagnosis
validate knowledge base
diagnose user requirements
diagnose old configuration

Figure 1 Configuration environment

Once the knowledge base is validated and deployed in productive use, there may be situations where no solutions can be computed for a set of actual customer requirements. In such situations, our configuration and diagnosis framework helps us to identify those parts of the user requirements prohibiting the successful configuration of the product. Finally, when given an already installed system and changed user requirements, the diagnosis techniques help us insofar that we are able *reconfigure* the system such that the new customer requirements can be satisfied.

Note that we defined a clear correspondence between the conceptual model and the logical representation. Therefore the inputs for the diagnostic reasoning can be mostly defined on the conceptual level, i.e., as UML diagrams containing configurations that are represented as instance diagrams. In addition, also the outcomes of the diagnostic reasoning can be traced back to those conceptual models.

The paper is organized as follows. Based on the modeling concepts presented in [8] we give an example for the construction of a conceptual PC product model. We extend the set of modeling concepts by introducing the notion of functional architectures from Mittal and Frayman [18] in order to explicitly design functional structures [2], which are relevant for the specification of (customer) requirements. In most cases customers are not interested in the detailed product topology but rather specify a set of functions the product must provide. In the line of [8] functional architectures are represented as UML stereotypes in the conceptual configuration model in the next section. Afterwards, we give a formal definition of a configuration task based on the definitions of [12] which is the formal foundation for knowledge acquisition and the integration of the diagnosis techniques. The final sections show how the different pieces of knowledge involved in the configuration task (product model, user requirements, and configuration results) can be analyzed using consistency-based diagnosis techniques. The paper ends with related work in the field and conclusions.

Concepts for modeling configuration knowledge bases

For presentation purposes we introduce a simplified (partial) UML model of a configurable personal computer (PC) as a working example. This model represents 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 are related to (customer) requirements, technical restrictions, economic factors, and restrictions according to the production process.

Figure 2 shows how UML is embedded into a four-layer architecture. The UML metamodel, i.e. the modeling concepts provided in UML are defined in MOF (Meta Object Facility) [5], which provides the concepts for designing metamodels in general. Using the concepts provided by UML, concrete schemas such as the configuration model in Figure 3 can be designed. A concrete configuration (result of the configuration process) represents an instance of a schema.

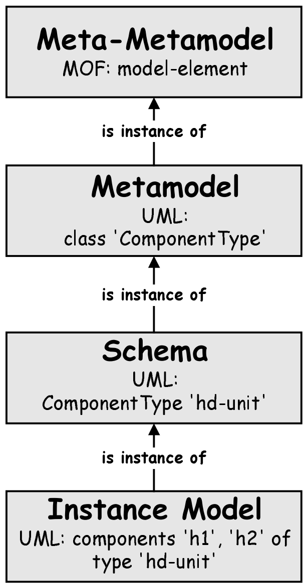


Figure 2 Integration of UML in a metamodel architecture

The basic means for defining additional modeling concepts inside UML is the introduction of a *profile*. A profile specializes the basic UML concepts (e.g. classes, associations, dependencies) for a specific domain by defining constraints on these concepts (definition of stereotypes). For the configuration domain we define special sets of classes *(component types, resource types, function types,* and *port types* are specializations of the UML concept *'class'),* associations *(incompatible, is_connected),* and dependencies *(requires, produces, consumes),* which are useful for designing configuration models.

In order to make the resulting configuration models executable, we propose a translation into the component port representation ([7], [12], [18]), which is well established for representing and solving configuration problems. The semantics of the different modeling concepts are formally defined by the mapping of the graphical notation to logical sentences based on the component port model. In general, consistency-based tools based on this component-port model can use the logic theory derived from the UML model although some transformation to the proprietary notation of a specific tool may have to be done. The following concepts are the basic parts of the ontology employed for designing configuration models (compare, e.g., [25]). Note, that we interpret ontologies in the sense of [3], i.e., ontologies are theories about the sorts of objects, properties of objects, and relations between objects that are possible in a specified domain of knowledge.

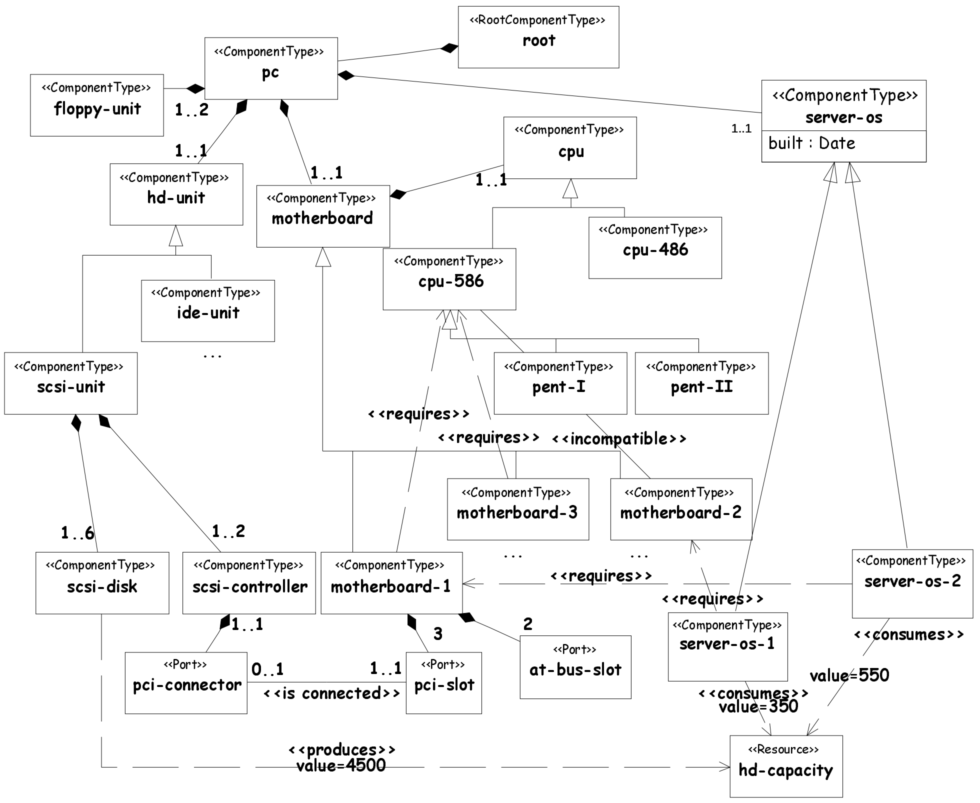
Component types. They represent parts the final product can be built of. Component types (e.g., component type *server-os in* Figure 3) are characterized by attributes that have a predefined domain of possible values.

Function types. They are used to model the functional architecture of an artifact. Similar to component types they can be characterized by attributes (see Figure 4).

Resources. Parts of a configuration problem can be seen as a resource-balancing task, where some of the component (function) types produce some resource and others are consumers (e.g. the *hd-capacity* is a resource produced by hard-disks and consumed by software units).

Generalization. Component (function) types with a similar structure are arranged in a generalization hierarchy (e.g. a *server-os* is either a *server-os-1* or a *server-os-2*).

Aggregation. Aggregations between components (functions) represented by part-of structures describe a range of how many subparts an aggregate can consist of (e.g. *cpu* is part of *motherboard*).



<<RootComponentType>> root
<<ComponentType>> pc
<<ComponentType>> floppy-unit
1..2
1..1
<<ComponentType>> server-os
built : Date
1..1
<<ComponentType>> cpu
1..1
<<ComponentType>> hd-unit
1..1
<<ComponentType>> motherboard
<<ComponentType>> cpu-486
<<ComponentType>> cpu-586
<<ComponentType>> ide-unit
...
<<ComponentType>> scsi-unit
<<ComponentType>> pent-I
<<ComponentType>> pent-II
<<requires>>
<<requires>>
<<incompatible>>
<<ComponentType>> motherboard-3
<<ComponentType>> motherboard-2
1..6
1..2
...
...
<<ComponentType>> scsi-disk
<<ComponentType>> scsi-controller
<<ComponentType>> motherboard-1
<<requires>>
<<ComponentType>> server-os-2
<<requires>>
1..1
3
2
<<ComponentType>> server-os-1
<<consumes>>
<<Port>> pci-connector
0..1
1..1
<<Port>> pci-slot
<<is connected>>
<<Port>> at-bus-slot
value=550
<<consumes>>
value=350
<<produces>>
value=4500
<<Resource>> hd-capacity

Figure 3 Conceptual product model

Connections and ports. In addition to the amount and types of the different components also the product topology may be of interest in a final configuration, i.e. how the components are interconnected with each other (e.g. a *pci-connector* is connected to a *pci-slot*).

Compatibility relations. Some types of components (functions) cannot be used together in the same final configuration, because they are *incompatible* (e.g. *motherboard-2* is incompatible with *cpu-586*). In other cases, the existence of one component (function) *requires* the existence of another special type in the configuration (e.g. *server-os-2 requires motherboard-1*).

Additional modeling concepts and constraints. Constraints on the product model, which cannot 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 [9].

As an example an OCL constraint on the product model could be:

```
context PC inv:
self.hd.isOclTypeOf(scsi-unit) implies
            self.floppy-unit->size = 1.
```

The constraint states that if the connected hard disk unit is of type *scsi-unit* then the number of connected floppies must be equal 1. The Object Constraint Language shows to be applicable to describe constraints on legal product constellations in a standardized declarative manner. The main features of the language are attribute access, *navigation* over associations (e.g., `self.floppy-unit`) resulting in collections of the connected instances, predefined operations on all types (`isOclTypeOf`), operations on collections (`size`), as well as logical (`implies`) and arithmetical operators.

The discussed modeling concepts have shown to cover a wide range of application areas for configuration [19]. Despite this, some application areas may have a need for special modeling concepts not covered so far. In order 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.

Functional configuration knowledge. Typical configuration knowledge incorporates a structural and a functional product architecture which are interrelated through a mapping from functions to components as described in [18]. While the structural architecture resembles partonomies and taxonomies of physical components, in many situations we want to have another more customer and functionality oriented *view* on the configurable artifact: The product is therefore modeled in terms of functionality or features that can be offered to and selected by the customer. Consequently, the language for the conceptual models is extended with capabilities (*function types*) to model this structure called the *functional architecture* [18]. Accordingly, the product can be described through a partonomy and taxonomy of functions that can again be characterized by attributes. The two views on the product (structural and functional) are interrelated through a "many-to-many" mapping between functions and physical components, whereby this mapping can be expressed through dependencies (requires) or additional constraints. Figure 4 shows the functional structure (with dimmed fill color) and the relations to the physical components of the structural model.

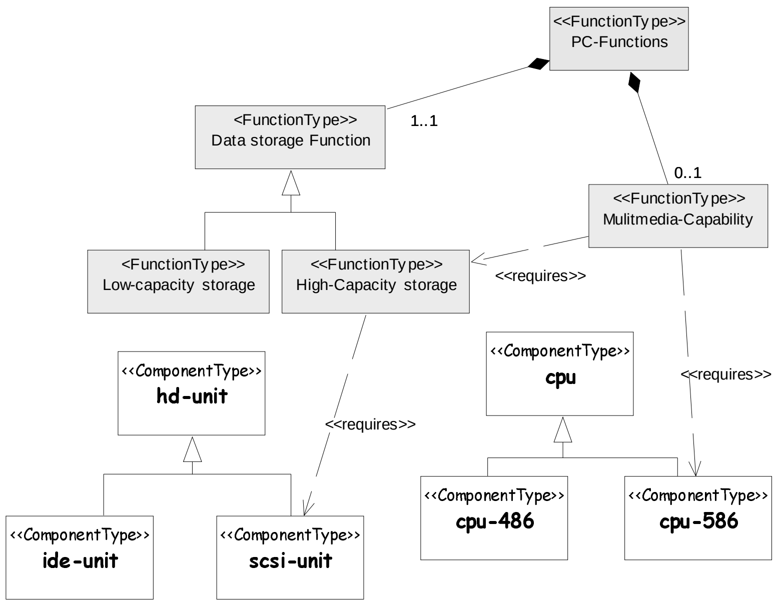


Figure 4 Functional product model

Structuring mechanisms. In case the conceptual models of the product (both structural and functional) are large and complex we utilize different mechanisms to cope with this complexity: First, the built-in *packaging* mechanism of the UML can be used to partition the model. Second, we can define different *views* (graphical depictions) on the underlying model beside the functional view, e.g., a connection-oriented view or a view on a certain substructure of the product. Finally, the usage of *contextual diagrams* can reduce the complexity of those complex product models (see [10]).

Automated knowledge base construction

Our framework of knowledge acquisition of configuration knowledge and the integration of diagnosis techniques relies on a general logical model of the configuration problem. This formalization allows us to define the semantics of the individual concepts of our modeling notation (UML) by giving translation rules from the conceptual level to the logic representation. In addition, the precise semantics that are provided by using first-order predicate logic as representation mechanism allows us to transform the conceptual models into the representation of other existing commercial configuration tools. Finally, basing on the analogy of the logic theory of configuration and a logic theory of the diagnosis task [21], we can easily integrate those techniques within our framework.

In practice, configurations are built from a predefined catalog of component types of a given application domain. Furthermore, the configuration task is characterized by a set of functional architectures, which specify the functional composition of the configurable artifact [12]. The set of function types can be seen as a special form of component types; from the viewpoint of the computation of configurations, the discrimination of function types and component types is not relevant. Component types as well as function types are described through a set of properties (attributes) with a predefined domain, and connection points (ports) representing logical or physical connections to other components.

We describe a configuration problem in terms of a set of logical sentences:

- A domain description (*DD*) containing information about the component types, their properties, ports and constraints on legal product constellations;
- a description of the specific user requirements (*SRS*), whereby the configuration problem has to be solved to conform to these requirements, and
- a set of predicate symbols (*CONL*) that can be used within the constraints from *DD* and are employed to describe configuration results. In our examples we will use the predicates *type/2, conn/4*, and *val/3* to describe the individual components, the attribute valuations and the connections between the component instances.

In our example, the predicate symbols in *CONL* are the *type*, *conn*, and *val* predicates. A fact *type(c,t)* assigns one of the given types to a component identification *c*, *conn(c1,p1,c2,p2)* represents a connection between components *c1* and *c2* via the ports *p1* and *p2*, and the fact *val(c,a,v)* describes the valuations of attribute *a* of a component *c* with the value *v*.

Definition (Configuration Problem): *In general we assume a configuration problem is described by a triple (DD, SRS, CONL) where DD and SRS are sets of logical sentences and CONL is a set of predicate symbols.*

DD represents the domain description (or configuration knowledge base), and SRS specifies the particular system requirements, which define an individual configuration problem instance. A configuration CONF is described by a set of positive ground literals whose predicate symbols are in the set CONL. ❑

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

These definitions allow for determining the consistency of configurations, but in order to ensure the completeness, we have to add specific completeness axioms for each predicate symbol in *CONL*, e.g.:

$type(X,Y) \Rightarrow \bigvee_{Z \in CONF} type(X,Y) = Z.$
$conn(V,W,X,Y) \Rightarrow \bigvee_{Z \in CONF} conn(V,W,X,Y) = Z.$
$val(V,A,C) \Rightarrow \bigvee_{Z \in CONF} val(C,A,V) = Z.$

We will denote a configuration CONF fulfilling these completeness axioms by $\overline{CONF}$.

Definition (Valid Configuration): *Let (DD, SRS, CONL) be a configuration problem. A configuration CONF is valid iff DD $\cup$ SRS $\cup$ $\overline{CONF}$ is satisfiable.* ❑

Furthermore, the domain description *DD* contains a set of application-independent axioms C_{Basic}, ensuring that e.g., connections are symmetric and one port can only be connected to one other port. For a formal exposition, see e.g., [12] or [7].

Having defined a logical model of configuration and given a notation for representing configuration knowledge on a conceptual level, we can derive the configurator knowledge base (*DD*) automatically by providing deterministic transformation rules.

The part of the knowledge base describing the product structure can be derived from the part-of structure of the conceptual model whereby the aggregate associations are mapped to the component-port representation. The available types, their attributes and connections are described by the *types*, *ports*, and *attrs* functions in *DD* (see below).

Furthermore, the additional constraints (like the *requires*) relations can be transformed and added to *DD*. As an example the constraint "*motherboard-2 incompatible with CPU-586*" from Figure 3 is represented in the domain description as named constraint:

Constraint C1: "cpu-586 incompatible motherboard-2"

type(ID1, CPU-586) $\wedge$ *type (ID2, motherboard-2)* $\wedge$
    conn(ID1, motherboard-port, ID2, cpu-port) $\Rightarrow$ *false.*

The next listing contains a part of the (generated) knowledge base for our PC example:

DD = {
    types = {pc, floppy-unit, hd-unit, scsi-unit, multimedia-
        capability, ... }.
    ports(pc)={floppy-unit-1, floppy-unit-2, hd-unit,
        motherboard}.
    ports(floppy-unit) = {pc}.
    ports(motherboard) = {pc,cpu}.
    attrs(server-os) = {built}.
    ...
  } $\cup$ C_{Basic} $\cup$ *{C1}*

For a detailed exposition on the transformation rules, see [8] and [9].

For the calculation of a configuration problem, the specific user requirements can be given in terms of a listing of key components or in terms of a set of additional constraints (logical sentences) that have to hold for the configuration. As an example, the user might specify the requirement of having a CPU of type *cpu-586* in the configuration by the following sentence:

SRS = { $\exists$ *P,M,C: type(P,pc)* $\wedge$ *type(C,cpu-586)* $\wedge$ *type(M,motherboard-1*
$\wedge$ *conn(P,motherboard,C,pc)* $\wedge$ *conn(C,motherboard,M,cpu))*
}

A configuration result *CONF* can be expressed as a set of ground literals using the predicates from *CONL*. A part of the configuration result for our example problem could be

CONF = {
type(p1, pc).
type(f1, floppy-unit).
type(f2, floppy-unit).
type(m1, motherboard-1).
type(c1, cpu-586).
type(s1, server-os-1).
...
conn(p1,floppy-unit-1,f1,pc).
conn(p1,floppy-unit-2,f2,pc).
conn(p1,motherboard,m1,pc).
conn(m1,cpu,c1,motherboard).
...
val(s1,built,"01-02-2000"). }

One additional important point of the automatic generation of the knowledge base (*DD*) is the tight correspondence to the original conceptual model by e.g., using the same names for component types and attributes and the naming for the generated constraints.

Note, that we can represent the both partial configurations, which may represent the user requirements in terms of *key components* [18] and the configuration result in terms of UML diagrams, i.e., configurations correspond to object diagrams (instance models, see Figure 2).

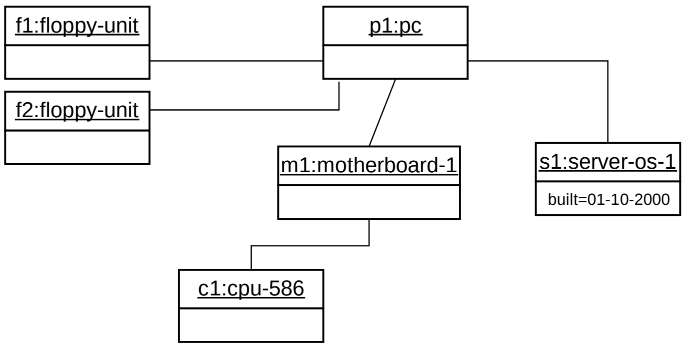


Figure 5 UML Instance diagram

Figure 5 shows a partial configuration expressed in terms of an UML object diagram. Using the transformation rules we can derive sets of logical sentences for *SRS* and *CONF* respectively, whereby this transformation is fairly straightforward because only simple *type*, *conn* and *val* facts have to be generated.

In the next section we will show how we can utilize consistency-based diagnosis techniques for validation and debugging purposes as well as a supporting mechanism during the deployment phase of the configuration system.

Diagnosing the knowledge base

In order to validate the knowledge base generated from the UML configuration model after the initial setup or after modifications, the domain expert or the knowledge engineer provides positive and negative examples, i.e., examples for configurations, which should be accepted by the knowledge base resp. not accepted by the knowledge base (see Figure 6). These example configurations play the role of "test cases" in standard Software Engineering validation processes. We denote the set of positive examples as E^+, the set of negative examples as E^-. All $e^+ \in E^+$ must be consistent with the knowledge base, all $e^- \in E^-$ must be inconsistent with the knowledge base. Note that examples can be partial configurations (e.g. some components, connections, or attributes are missing) and complete configurations as well.

If some e^+ are inconsistent with the knowledge base, the question must be answered, which set of constraints must be eliminated for making the knowledge base accepting those e^+. Additionally we have to find an extension *EX* such that the knowledge base does not accept any $e^- \in E^-$.

The two example sets serve complementary purposes. The goal of the positive examples in E^+ is to check that the knowledge base will accept correct configurations; if it does not, i.e. a particular positive example e^+ leads to an inconsistency, we know that the knowledge base currently is too restrictive. Conversely, a negative example serves for checking the restrictiveness of the knowledge base; negative examples correspond to real-world cases that are configured incorrectly, and therefore a negative example that is accepted means that a relevant condition is missing in the knowledge base.

In the line of consistency-based diagnosis, an inconsistency between *DD* and the positive examples means that a diagnosis corresponds to the removal of possibly faulty sentences from *DD* such that the consistency is restored. Conversely, if that removal leads to a negative example e^- becoming consistent with the knowledge base, we have to find an extension that, when added to *DD*, restores the *inconsistency* for all such e^-. Figure 6 shows an overview of the approach of consistency-based diagnosis of the configurator knowledge base.

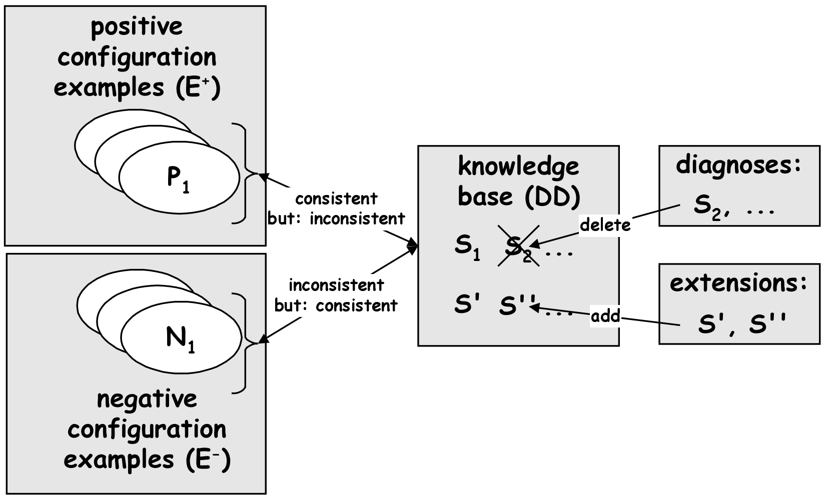

positive configuration examples (E^+)
P_1
consistent but: inconsistent
inconsistent but: consistent
N_1
negative configuration examples (E^-)
knowledge base (DD)
S_1 S_2 ...
S' S'' ...
delete
add
diagnoses:
S_2, ...
extensions:
S', S''

Figure 6 Consistency-based diagnosis of the knowledge base

Definition (CKB Diagnosis Problem): *A CKB-Diagnosis Problem (Diagnosis Problem for a Configuration Knowledge Base) is a triple (DD, E^+, E^-) where DD is a configuration knowledge base, E^+ is a set of positive and E^- a set of negative examples. The examples are given as sets of logical sentences. We assume that each example on its own is consistent.* ❑

Definition (CKB Diagnosis): *A CKB diagnosis for a CKB-Diagnosis Problem (DD, E^+, E^-) is a set $S \subseteq DD$ of sentences such that there exists an extension EX, where EX is a set of logical sentences, such that*

$$DD - S \cup EX \cup e^- \text{ is consistent } \forall e^+ \in E^+,$$
$$DD - S \cup EX \cup e^- \text{ inconsistent } \forall e^- \in E^-. \square$$

A diagnosis *S* will always exist under the reasonable assumption that the positive and the negative examples do not interfere with each other.

Proposition: *Given a CKB-Diagnosis Problem (DD, E^+, E^-), a diagnosis S for (DD, E^+, E^-) exists iff*

$$\forall e^+ \in E^+ : e^+ \cup \bigwedge_{e^- \in E^-} (\neg e^-) \text{ is consistent. } \square$$

From here on, we will refer to the conjunction of negated negative examples as **NE**, i.e., $NE = \bigwedge_{e^- \in E^-} (\neg e^-)$

The last proposition however, lets us find a characterization of diagnoses without the explicit specification of these extensions.

Corollary: *S is a diagnosis iff $\forall e^+ \in E^+ : DD - S \cup e^+ \cup NE$ is consistent.*

For the computation of such diagnoses (explanations of unexpected behavior of the configurator) we can adapt the standard hitting set algorithm from model-based diagnosis [21]. For focusing purposes, the concept of *conflict sets* is defined as follows:

Definition (Conflict set): *A conflict set CS for (DD, E^+, E^-) is a set of elements of DD such that $\exists e^+ \in E$: $CS \cup e^+ \cup NE$ is inconsistent. We say that, if $e^+ \in E^+ : CS \cup e^+ \cup NE$ is inconsistent, that e^+ induces CS.* ❑

For the collection *F* of conflict sets for *(DD, E^+, E^-)* a directed acyclic graph for the computation of the minimal hitting sets (*HS-DAG*) is constructed in breadth-first manner [21]. The nodes are labeled with conflict sets from *F* (containing conflicting constraints), edges leading away are labeled with elements from the conflict set of the node. At each node, the theorem prover (in our case the configuration engine) is called to test whether the positive examples are consistent with the examples and can be completed to working configurations, if we eliminate all constraints on the path from the root node to the current node from the knowledge base (*DD*) and add the negated negative examples. The breadth-first construction leads to the effect that diagnosis are computed in order of their cardinality and additional tree pruning techniques can be employed to reduce the search complexity.

For a detailed exposition of the computation of CKB-diagnoses, see [7].

The outcome of the diagnostic processes is a set of diagnoses that explain the unexpected behavior of the configurator. Each diagnosis contains a set of constraints from *DD* that have possibly to be revised (or to be canceled) in order to repair the knowledge base.

Note, that these constraints from *DD* were automatically generated and were given appropriate names. Therefore, these results are strongly related to the original conceptual product model. One can imagine that when a tool for graphical knowledge acquisition is used, we could highlight those faulty chunks of knowledge in the graphical depiction and repair can be done on this level in the best case. At least, the name and the description of the involved constraints, e.g., *"Constraint C1 : cpu-586 incompatible motherboard-2"* can be returned to the test engineer as a hint where to focus his/her debugging efforts.

For the definition of the positive and negative examples we can also use instance diagrams that represent partial or complete configurations. Consequently, these examples are easy to understand and communicate to domain experts. Finally, the positive examples do not need to be specified by hand but can be existing configurations from former configuration runs that still have to be consistent with the altered knowledge base.

Diagnosis of requirements and reconfiguration support

Diagnosing user requirements

Even once the knowledge base has been tested and corrected, diagnosis techniques can play a significant role in the configuration process. Instead of an engineer testing an altered (extended or updated) knowledge base, we are now dealing with an end user (customer or sales representative) who is using the tested knowledge base for configuring actual products. Such users frequently face the problem of requirements being inconsistent because they exceed the feasible capabilities of the system to be configured. In such a situation, the diagnosis approach presented here can now support the user in finding which of his/her requirements produces the inconsistency. Formally, the altered situation can be easily accommodated by swapping requirements and domain description in the definition of CKB diagnosis. Formerly, we were interested in finding particular sentences from *DD* that contradicted the set of examples. Now we have the user's system requirements *SRS*, which contradict the domain description (see Figure 7). The domain description is used in the role of an all-encompassing partial example for correct configurations.

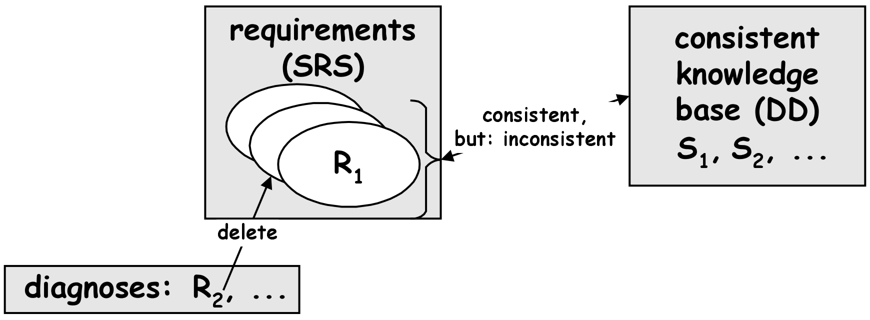

requirements (SRS)
R1
consistent, but: inconsistent
consistent knowledge base (DD) S1, S2, ...
delete
diagnoses: R2, ...

Figure 7 Diagnosing user requirements

Definition (CREQ Diagnosis Problem): *A configuration requirements diagnosis (CREQ-Diagnosis) problem is a tuple (SRS,DD), where SRS is a set of system requirements and DD a configuration domain description. A CREQ Diagnosis is a subset $S \subseteq SRS$ such that $SRS - S \cup DD$ is consistent.* ❑

Definition (CREQ Conflict Set): *A conflict set $CS \subseteq SRS$ for (SRS, DD) is a set, such that $CS \cup DD$ is inconsistent.* ❑

Reconfiguration support

There is an increasing demand for software supporting after-sales activities in various application domains [16]. Especially the need for supporting reconfiguration of product individuals, i.e., the modification of a concrete product instance in order to meet the new requirements, is an open research area. These new requirements arise e.g., when a customer wants to upgrade the existing system to provide new or altered functionality or when parts of the existing system are broken and have to be replaced by a newer version of that component, because the original components are no more available.

When comparing reconfiguration with configuration, the main difference lies in an existing product which mainly influences the process of reconfiguration. Given that the customer requirements have changed (or there are new regulations defined in the knowledge base), the goal of the reconfiguration process is to compute a configuration, where most parts of the existing configuration can be preserved, i.e., the number of needed changes (changes of parameters and connections or removal of components) is minimized. Beside the minimization of the number of needed changes the reconfiguration process can be guided by other optimization functions, e.g., the different types of changes may be associated with different costs, i.e., a change in the parameters may be cheaper than exchanging a whole component. Finally, an optimization criterion may lie in the number (and the associated costs) of the components that have to be added in order to provide the additional functionality.

Within our framework we can utilize diagnosis techniques to support the reconfiguration process as follows: Typically, a description of the existing configuration (CONF) is given that is not consistent with the altered user requirements (SRS) and the domain description (DD). Reconfiguration is performed as follows: If we cannot extend the existing configuration to cope with the new requirements, we search for a set of parts of the existing configuration that have to be removed (or exchanged) such that the remaining existing configuration can be completed in a way the new requirements are fulfilled. Since we are trying to find "minimal" or "suitable" (with respect to the objective function) sets of parts to be removed, we have the effect that in the reconfiguration process we try to preserve most of the existing system. Obviously, we could start a new configuration from scratch with the new requirements, but this would possibly lead to a complete different configuration where none of the existing parts can be reused.

Figure 8 depicts the integration of diagnosis for the reconfiguration support schematically.

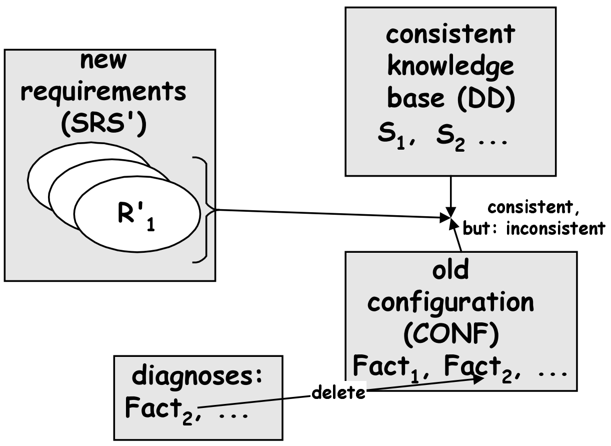


Figure 8 Diagnosis for reconfiguration

Within our logical framework for configuration and diagnosis, we can define the reconfiguration problem as follows.

Definition (Reconfiguration Problem): *A reconfiguration problem is a quadruple (DD, SRS, CONL, CONF), where (DD, SRS, CONL) is a configuration problem, whereby SRS contains the altered requirements, and CONF represents the existing configuration using the predicates from CONL. A reconfiguration problem arises if DD $\cup$ SRS $\cup$ CONF is inconsistent.* ❑

Note, that *SRS* may not only contain the new customer requirements but in addition those parts of the existing configuration that should definitely not be changed. This situation may be given, if it is known before the reconfiguration process, that the exchange of certain parts will definitely not result in a suitable reconfiguration, e.g., exchanging some key parts which would cause too high costs.

We define the concept of *Reconfiguration Diagnosis* and *Reconfiguration Solution* as follows.

Definition (Reconfiguration Diagnosis): *Given a Reconfiguration Problem RP (DD, SRS, CONL, CONF), a Reconfiguration Diagnosis is a set S* $\subseteq$ *CONF, such that* $CONF - S \cup DD \cup SRS$ *is consistent.*

Every solution to the configuration problem (DD, SRS $\cup$ *(CONF* $-$ *S), CONL) is a reconfiguration solution for RP.* ❑

Given this definition, in principle every solution to the configuration problem (DD, SRS, CONL) is a valid reconfiguration solution. Therefore, a reconfiguration solution will always exist under the assumption that in the worst case all elements from *CONF* are contained in the reconfiguration diagnosis.

For the computation of reconfiguration diagnosis, we can again use the adapted hitting set algorithm from [21]. Conflict sets are (minimal) subsets from *CONF* that together with $DD \cup SRS$ cause a contradiction. During the HS-DAG construction it is tested whether the remaining parts of the existing configuration can be completed to a working configuration that satisfies the user requirements. Depending on the application domain, each time a reconfiguration diagnosis is identified we can store one, all, or the "optimal" reconfiguration solution (with respect to some objective function) and apply e.g., a branch-and-bound optimization algorithm.

In many application domains of configuration where the products are very complex (e.g., telecommunication switches [11]) it is not always feasible to compute "optimal" solutions, but the goal is to come up (possibly guided by heuristics) with a "good" or "suitable" solution. Accordingly, when searching for reconfiguration alternatives the user may be satisfied, when a set of "suitable" alternative configurations is calculated. The search space for reconfiguration alternatives can be reduced by limiting the search depth or by reducing the search for reconfiguration diagnosis to parameters and connections; Furthermore we assume that in typical reconfiguration problems only a small part of the existing configuration has to be revised, i.e., the cardinality of the diagnoses is rather small compared to the size of *CONF*. However, in cases the reconfiguration diagnoses will have a higher cardinality, additional techniques like hierarchical abstraction to reduce the search complexity may be applied. Finally, we can employ a heuristics-driven search algorithm where the diagnoses are not generated in order of their cardinality which is the effect of the breadth-first construction of the HS-DAG in [21]. Having in mind that in many cases we are only interested in *suitable* diagnoses and solutions, the side effect of having non-minimal diagnoses will be tolerable, since these solutions can be computed efficiently.

Related work

The formalization of the semantics of conceptual design languages like UML is an actual research topic. Automated generation of logic-based descriptions through the translation of domain-specific modeling concepts expressed using the concepts of a standard design language has not been discussed so far. The focus of automated and knowledge-based software engineering [15] is automated software reuse, where program construction is realized reusing existing software libraries. In [1] a formal semantics for object model diagrams based on OMT [23] is defined in order to support the assessment of requirement specifications.

Architecture description languages (ADLs) provide basic concepts for software development focused on one or more high level models of the system (components, external systems, source modules etc.). The major challenge in this area is to integrate proprietary ADLs into the industrial development process. In [22], an integration of two ADLs *(C2 and Wright)* is proposed using the extension mechanisms provided in UML. We view our work as complementary since our goal is the generation of executable logic descriptions.

An overview on aspects and applications of functional representations is given in [2], where the functional representation of a device is divided into three parts: the intended function, the structure of the device, and a description of how a device achieves a function (represented through a process description). Mittal and Frayman [18] propose the integration of functional architectures into the configuration model by defining a matching from functions to key components, which must be part of the configuration if the function should be provided. This interpretation for the achievement of functions is used in our framework.

There is a broad spectrum of representation formalisms employed in knowledge-based configuration systems [26]. The increasing complexity of configuration knowledge bases demands the provision of advanced concepts supporting the knowledge acquisition task. In this paper we have shown how to employ a standard design language (UML) in order to design executable conceptual configuration models. Furthermore, the integration of conceptual modeling techniques and model-based diagnosis techniques tackles open issues in the area of development environments for knowledge-based configuration systems.

Model-based diagnosis techniques were initially developed to identify faults in physical devices, e.g., electronic circuits. Later on, these techniques were also applied for diagnosis and debugging of software systems. The application fields of software diagnosis range from diagnosis of logic programs using expected and unexpected query results to identify incorrect clauses [4], diagnosis of constraint violations in databases [14], or diagnosis of hardware designs defined in VHDL [13]. Our work of diagnosing configurator knowledge bases is in the line of software diagnosis, where test cases (examples) are used to validate the software system.

Crow and Rushby [6] extend Reiter's [21] framework of model-based diagnosis in order to integrate *repair* (reconfiguration) functionality, claiming that the real goal of the diagnosis task is not only to detect causes for unexpected behavior but also to repair the system. Following a diagnose-and-repair approach they first diagnose the faulty behavior and then try to *reconfigure* the system in order to re-establish the desired functionality. Their approach mainly relies on the notion of having pre-enumerated spare parts or redundant elements within a system, and they associate a special predicate for each component that describes if the component is reconfigured or not and what the effects of the reconfiguration are. After identification of causes of the unexpected behavior, a consistency-based approach is employed in order to find a (minimal) set of components that have to be reconfigured in order to re-establish the system's functionality. In their work, this reconfiguration knowledge has to be explicitly modeled within the knowledge base, which causes additional knowledge acquisition and maintenance efforts. However, although the search space for reconfiguration alternatives is reduced, their approach does not in general apply to the area of reconfiguration of products in our sense, because in our domain the set of alternative or newly added components cannot be pre-enumerated.

Männistö et al. [16] propose a reconfiguration approach, where reconfiguration knowledge is explicitly represented through a set reconfiguration operations, where an optimal reconfiguration can be calculated by evaluating the generated reconfiguration sequence. In the approach described in [16], the knowledge that is needed to configure product individuals from scratch is enriched with additional reconfiguration knowledge. This explicit reconfiguration knowledge consists of two basic parts. First, a set of reconfiguration operations defining possible changes to the existing product, whereby these operations consist of a *precondition* and an *action*. Typical actions include addition or removal of parts. Second, a set of reconfiguration invariants that have to hold are defined as the other part of the reconfiguration knowledge. Different alternatives for reconfiguring the system can be discriminated according to a *value* function that takes the needed change operations and the resulting reconfigured system into account. This conceptual model for reconfiguration is related to our model of reconfiguration in terms of being independent from a special problem solving mechanism. However, compared to the model-based reconfiguration model described in previous sections, the reconfiguration knowledge is modeled explicitly and only the pre-defined reconfiguration actions can be performed.

Conclusions

In this paper we have shown how to employ techniques from software engineering and knowledge-based systems for the design of knowledge-based configuration systems. With the increasing size and complexity of knowledge bases the usefulness of these techniques is likewise growing. We have presented an approach for representing configuration knowledge bases on a conceptual level where the resulting models are automatically translated into a representation formalism widely used in the configuration domain. 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 reduce the development time and effort significantly because these high-level descriptions are directly executable. Second, standard design techniques like the UML are far more comprehensible and are widely adopted in the industrial software development process.

In order to support the validation of configuration knowledge bases as well as the diagnosis of unfeasible (customer) requirements and old configurations we have proposed the application of model-based diagnosis techniques. In particular, due to its conceptual similarity to configuration, consistency-based diagnosis is a highly suitable technique to aid in the debugging of configurators. The proposed definition enables us to clearly identify the causes that explain misbehavior of the configurator and the unfeasibility of (customer) requirements. Furthermore, we can utilize the same techniques to support the reconfiguration process.

The concepts presented in this paper are currently implemented in a prototype configuration environment. For the knowledge acquisition phase, any CASE tool supporting the UML can be used because the resulting models can be stored in a standardized XML representation. The conceptual product models are then translated (according to the defined semantics) into the representation of a constraint-based commercial configuration tool (ILOG Configurator [17]) that is used to solve the actual configuration task. Finally, we have implemented a general diagnosis component that interacts with the configurator software and can be used to diagnose the knowledge base, the user requirements and existing configurations as described in this paper.

References

[1] Bourdeau, R.H., Cheng, B.H.C.: A Formal Semantics for Object Model Diagrams, IEEE Transactions on Software Engineering, 1995, Vol. 21, No. 10, pp. 799-821.

[2] Chandrasekaran, B., Goel, A., Iwasaki, Y.: Functional Representation as Design Rationale. IEEE Computer, Special Issue on Concurrent Engineering, 1993, pp. 48-56.

[3] Chandrasekaran, B., Josephson, J., and Benjamins, R.: What are ontologies, and why do we need them? IEEE Intelligent Systems, 1999, 14(1), pp. 20-26.

[4] Console, L., Friedrich, G., and Dupré, D.T.: Model-based diagnosis meets error diagnosis in logic programs. In Proceedings International Joint Conference on Artificial Intelligence, Chambery, Morgan Kaufmann, August 1993, pp. 1494-1501.

[5] Crawley, S., Davis, S., Indulska, J., McBride, S., Raymond, K.: Meta Information Management, In Proceedings 2nd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'97), Canterbury, UK, July, 1997.

[6] Crow, J., Rushby, J.: Model-Based Reconfiguration: Toward an Integration with Diagnosis, Proceedings National Conference on Artificial Intelligence AAAI, 1991, Vol.2, pp. 836-841.

[7] Felfernig, A., Friedrich, G., Jannach, D., and Stumptner, M.: Consistency-based diagnosis of configuration knowledge bases, In Proceedings 14th European Conference on Artificial Intelligence (ECAI'2000), Berlin, Germany, 2000, pp. 146-150.

[8] Felfernig, A., Friedrich, G., and Jannach , D., UML as domain specific language for the construction of knowledge-based configuration systems, In: International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Vol. 10(4), 2000, pp. 449-470.

[9] Felfernig, A., Friedrich, G., and Jannach , D., Generating product configuration knowledge bases from precise domain extended UML models, 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'00), Chicago, USA, 2000, pp. 284-293.

[10] Felfernig, A., Jannach, D., and Zanker, M.: Contextual diagrams as structuring mechanism for designing configuration knowledge bases defined in UML, Proceedings 3rd Intl. Conference on the Unified Modeling Language (*UML'2000*), York, UK, 2000, pp. 240-254.

[11] Fleischanderl G., Friedrich, G., Haselböck, A., Schreiner, H., and Stumptner, M.: Configuring Large Systems Using Generative Constraint Satisfaction. IEEE Intelligent Systems, Vol. 13(4), July/August 1998, pp. 59–68.

[12] Friedrich, G. and Stumptner, M.: Consistency-Based Configuration, AAAI'99 Workshop on Configuration, AAAI Press Technical Report WS-99-05, Orlando, Florida, 1999, pp. 35-40.

[13] Friedrich, G., Stumptner, M., and Wotawa, F.: Model-Based Diagnosis of Hardware Designs, in: Artificial Intelligence, Vol. 111(2), 1999, pp 3-39.

[14] Gertz, M. and Lipeck, U.W.: A Diagnostic Approach to Repairing Constraint Violations in Databases. In Proceedings DX'95 Workshop on Principles of Diagnosis, Goslar, October 1995.

[15] Lowry, M., Philpot, A., Pressburger, T., and Underwood, I., A Formal Approach to Domain-Oriented Software Design Environments, in Proc. 9th Knowledge-Based Software Engineering Conference, Monterey, CA, September 1994, pp. 48-57.

[16] Männistö, T., Soininen, T., Tiihonen, J., and Sulonen, R.: Framework and Conceptual Model for Reconfiguration, AAAI Workshop on Configuration, AAAI Press Technical Report WS-99-05, Orlando, Florida, 1999, pp. 59-64.

[17] Mailharro, D.: A classification and constraint-based framework for configuration, AIEDAM, special issue: Configuration Design, Vol. 12(4), 1998, pp. 383-397.

[18] Mittal, S. and Frayman, F.: Towards a generic model of configuration tasks, Proc. Intl. Joint Conference on Artificial Intelligence (IJCAI'89), Detroit, Morgan Kaufmann, 1989, pp. 1395-1401.

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

[20] Pine II, B.J., Victor, B., Boynton, A.C.: Making Mass Customization Work. Harvard Business Review, Sep./Oct. 1993, pp. 109-119.

[21] Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence, Vol. 32(1), 1987, pp. 57–95.

[22] Robbins, J.E., Medvidovic, N., Redmiles, D.F., and Rosenblum, D.S.: Integrating Architecture Description Languages with a Standard Design Method, Proc. 20th Intl. Conference on Software Engineering, Kyoto, Japan, 1998, pp. 209-218.

[23] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall International Editions, New Jersey, 1991.

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

[25] Soininen, T., Tiihonen, J., Männistö, T., and Sulonen, R.: Towards a general ontology of configuration, AIEDAM, special issue: Configuration Design, Vol. 12(4), 1998, pp. 357-372.

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