CONCEPTUAL DESIGN FOR CONFIGURATION AND RECONFIGURATION OF CUSTOMIZABLE PRODUCTS

A. FELFERNIG, G. FRIEDRICH, AND D. JANNACH
Institute for Computer Science and Manufacturing
University of Klagenfurt, Austria

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 configurator development time is strictly limited, i.e. 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) in order to represent the configuration knowledge on a conceptual level. The two constituent parts of the conceptual configuration model are the component model and a set of corresponding functional architectures defining which requirements can be imposed to the product. The conceptual configuration model is translated into an executable logic representation. Using this representation we show how to employ model-based diagnosis for diagnosing a faulty configuration knowledge base, infeasible requirements, and for reconfiguring old configurations.

1. Introduction

Product configuration systems play an important role in the support of the mass customization paradigm (Pine, Victor, and Boynton 1993). 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. 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 (Felfernig, Friedrich, and Jannach 1999). UML (Rumbaugh, Jacobson, Booch 1998) is a conceptual modeling language that is widely applied in industrial software development processes. This notation is similar to OMT diagrams (Object Modeling

Technique) (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991) 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, the resulting configuration knowledge base is employed in productive use. Given the actual (customer) requirements the configuration system calculates corresponding solutions. Finally, old configurations must be reconfigured in order to be consistent with the new (customer) requirements. The diagnosis component is the basis for all three processes (knowledge acquisition, diagnosis of requirements, and diagnosis of obsolete components in old configurations).

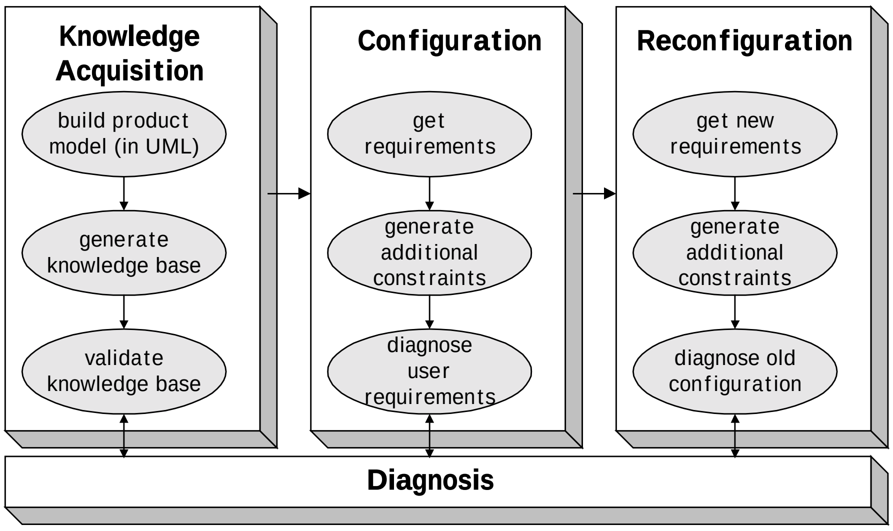

Knowledge Acquisition
build product model (in UML)
generate knowledge base
validate knowledge base
Configuration
get requirements
generate additional constraints
diagnose user requirements
Reconfiguration
get new requirements
generate additional constraints
diagnose old configuration
Diagnosis

Figure 1: Configuration environment

Based on the modeling concepts presented in (Felfernig, Friedrich, and Jannach 1999) 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 (Mittal and Frayman 1989) in order to explicitly design functional structures (Chandrasekaran, Goel, and Iwasaki, 1993), which are relevant for the specification of (customer) requirements. Mostly, customers are not interested in the detailed product topology but rather specify a set of functions the product must provide. In the line of (Felfernig, Friedrich, and Jannach 1999) functional architectures are represented as well as UML stereotypes in the conceptual configuration model (Section 2). In Section 3 we give a formal definition of a configuration task which extends the definition of (Friedrich and Stumptner 1999) by explicitly considering functions in the logic representation. In

Section 4 we recall the notion of consistency-based diagnosis of configuration knowledge bases and consistency-based diagnosis of (customer) requirements (Felfernig, Friedrich, Jannach, and Stumptner 2000). Based on these concepts we employ model-based diagnosis in reconfiguration (Section 5). Finally, Sections 6 and 7 contain related work and conclusions.

2. Concepts for designing configuration models

For presentation purposes we introduce a simplified (partial) UML model of a configurable PC (Figure 3) as a working example. This diagram 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) (Crawley, Davis, Indulska, McBride, Raymond, 1997), 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.

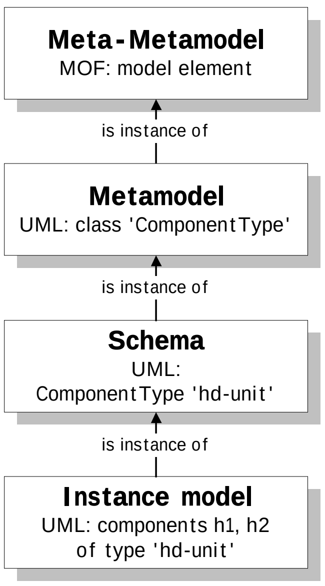


Figure 2: Integration of UML in a metamodel architecture

The basic means for defining additional modeling concepts in 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 class), associations (*is connected*), and dependencies (*requires, incompatible, 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 (Mittal and Frayman 1989), 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 (see Section 3). In general, consistency-based tools based on this model can use the logic theory derived from the UML configuration model. The following concepts are the basic parts of the ontology[1] employed for designing configuration models (Soininen, Tiihonen, Männistö, and Sulonen 1998).

Component types These represent parts the final product can be built of. Component types (e.g. component type *server-os-1 in* Figure 3) are characterized by attributes.

Function types They are used to model the functional architecture of an artifact. Similar to component types they can be characterized by attributes (e.g. *hd-function* in 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*).

[1] We interpret ontologies in the sense of (Chandrasekaran, Josephson, and Benjamins 1999), 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.

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

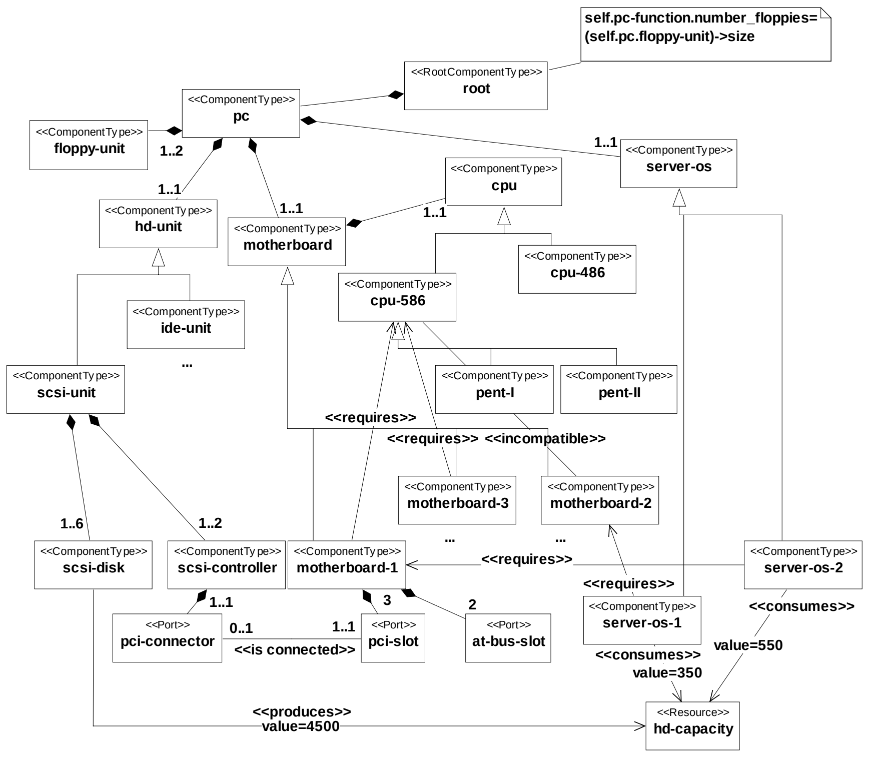

self.pc-function.number_floppies=
(self.pc.floppy-unit)->size
<<RootComponentType>> root
<<ComponentType>> pc
<<ComponentType>> floppy-unit
<<ComponentType>> server-os
<<ComponentType>> cpu
<<ComponentType>> hd-unit
<<ComponentType>> motherboard
<<ComponentType>> cpu-486
<<ComponentType>> cpu-586
<<ComponentType>>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

Figure 3: Conceptual configuration 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 and requirements relations Some types of components (functions) cannot be used together in the same final configuration, 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*).

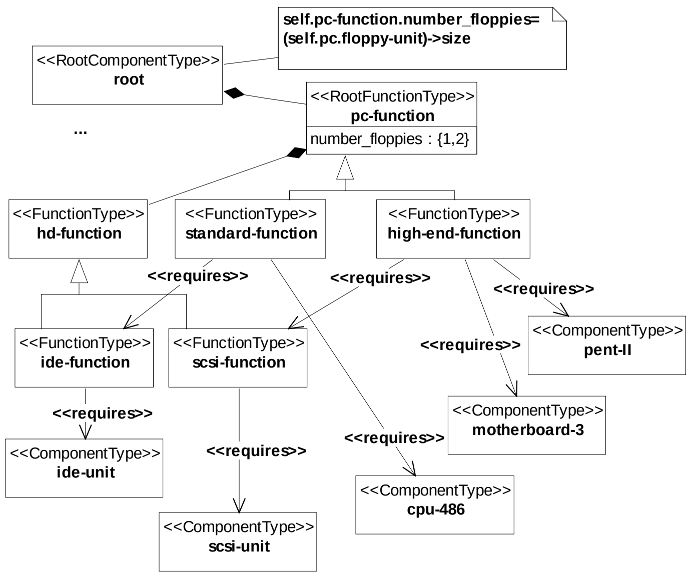

self.pc-function.number_floppies=
(self.pc.floppy-unit)->size
<<RootComponentType>> root
...
<<RootFunctionType>> pc-function
number_floppies : {1,2}
<<FunctionType>> hd-function
<<FunctionType>> standard-function
<<FunctionType>> high-end-function
<<requires>>
<<FunctionType>> ide-function
<<FunctionType>> scsi-function
<<ComponentType>> pent-II
<<ComponentType>> motherboard-3
<<ComponentType>> ide-unit
<<ComponentType>> cpu-486
<<ComponentType>> scsi-unit

Figure 4: Functional architecture pc-function

Additional modeling concepts and constraints Constraints on the product model, which can not be expressed graphically, are formulated using the language OCL (Object Constraint Language), which is an integral part of UML. As it is done for the graphical modeling concepts, OCL expressions are translated into a logical representation executable by the configuration engine. The discussed modeling concepts have shown to cover a wide range of application areas for configuration (Peltonen, Männistö, Soininen, Tiihonen, Martio, and Sulonen 1998). Despite this, some application areas may have a need for special modeling concepts not covered so far. To introduce a new modeling concept a new stereotype has to be defined. Its semantics for the configuration domain must be defined by stating the facts and constraints induced to the logic theory when using the concept.
In order to explicitly model the structure of (customer) requirements we introduce the concept of *functional architectures*.

Definition (Functional Architecture): Let *FA* be the root function type (stereotype <<*RootFunctionType*>>) of the configuration model, which is a direct part of the root component type (stereotype <<*RootComponentType*>>) of the configuration model. Then *FA* is a *functional architecture*, which includes the *function types* which are directly or transitivly connected with *FA* via generalizations or aggregations, the *corresponding attributes*, and connected *part-of relations*. Furthermore, all *constraints* exclusively concerning functions and attributes of *FA*, belong to *FA*.❑

The *pc-function* (Figure 4) represents a functional architecture in the PC configuration model. The mapping from functions to components is modeled using the requires relations in the simple case. More complex relationships between functions and components can either be represented by additionally defined modeling concepts or OCL constraints.

The OCL constraint

```
self.pc-function.number_floppies = (self.pc.floppy-unit)->size
```

assigned to *root* (root component type of the configuration model) assures, that the number of floppies in the component model is equal to the number of required floppies (*number_floppies*).

In the following we give a definition of a configuration task, which includes the definition of functional architectures. Based on this logical foundation we sketch the automatic translation of the conceptual model into this logical representation.

3. Definition of a configuration task

The following definition of a configuration task extends the definition given in (Friedrich and Stumptner 1999) by considering functions as part of the configuration task. In practice, configurations are built from a predefined catalog of component types (*types*) of a given application domain. Furthermore, the configuration task is characterized by a set of functional architectures which specify the functional composition of artifacts and constraints on their composition, i.e. a set of necessary and optional functions, and constraints on their composition (Friedrich and Stumptner 1998), (Mittal and Frayman 1989). The set of function types is further denoted as *functions*. Component types as well as function types are described through a set of properties (*attributes*), and connection points (*ports*) representing logical or physical connections to other components. Both, attributes and ports have an assigned domain (*dom*).

The domain description (*DD*) of a configuration task contains this information (*types, functions, ports, attributes, dom*) and additional constraints on legal configurations. The actual configuration task has to be solved according to the set *SRS* (system requirements specification).

The *DD* is derived by translating the component structure as well as the functional architecture(s) and the corresponding constraints. The *DD* for the PC configuration model (Figure 3, Figure 4) is the following.

```
types={root, pc, floppy-unit, hd-unit, motherboard, cpu, cpu-586,...}.
functions={pc-function, hd-function, ide-function, scsi-function,...}.
attributes(server-os-1)={version}.
attributes(pc-function)={number_floppies}.
...
dom(pc-function, number_floppies)={1,2}.
...
ports(root)={pc-port, pc-function-port}.
ports(pc)={root-port, floppy-unit-port, hd-unit-port, motherboard-port...}.
ports(pc-function)={root-port, hd-function-port-1, ...}.
...
```

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

- ***FUNCS*** is a set of literals of the form *func(c,t)*. *t* is included in the set of *functions* defined in *DD*. The constant *c* represents the identification for a function.
- ***COMPS*** is a set of literals of the form *type(c,t)*. *t* is included in the set of *types* defined in *DD*. The constant *c* represents the identification for a component.
- ***CONNS*** is a set of literals of the form *conn(c1,p1,c2,p2)*. *c1* and *c2* are component identifications from COMPS, *p1* (*p2*) is a port of the component *c1* (*c2*).
- ***ATTRS*** is a set of literals of the form *val(c,a,v)*, where *c* is a component-identification, *a* is an attribute of that component, and *v* is the actual value of the attribute (selected out of the domain of the attribute)[2].

Example for a configuration result:

```
type(r1,root).
func(fp1,standard-function).
func(fhd1,ide-function).
val(fp1,number_floppies,1).
type(p1,pc).
type(f1, floppy-unit).
type(m1,motherboard-2).
type(c1,cpu-486).
...
conn(fp1,root-port,r1,pc-function-port).
conn(fhd1,pc-function-port,fp1,hd-function-port).
conn(p1,root-port,r1,pc-port).
conn(m1,pc-port,p1,motherboard-port).
conn(c1,motherboard-port,m1,cpu-port).
conn(f1, pc-port, p1, floppy-unit-port-1).
...
```

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.

[2] If not explicitly mentioned, variables are all-quantified.

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 *FUNCS*, *COMPS*, *CONNS*, and ATTRS represent a configuration result, then the configuration is consistent exactly *iff DD* ∪ *SRS* ∪ *FUNCS* ∪ *COMPS* ∪ *CONNS* ∪ *ATTRS* can be satisfied. ❑

Additionally we have to specify that that *FUNCS* includes all required functions, *COMPS* includes all required components, *CONNS* describes all required connections, and *ATTRS* includes a complete value assignment to all variables in order to achieve a *complete* 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 (Friedrich and Stumptner, 1999).

In order to translate the graphical constraints we provide a set of translation rules, which will not be discussed in this paper[3]. E.g. the *requires* relation between the *standard-function* and the *ide-function* (Figure 4) is translated into the following logical sentence.

```
type(ID1, standard-function) ∧ type(ID2, root) ∧ conn(ID1, root-port, ID2,
                              standard-function-port)
∃ ID3 type(ID3, ide-function) ∧ conn(ID3, root-port, ID2, hd-function-port).
```

The constraint *motherboard-2 incompatible* with *CPU-586* (Figure 3) is translated as follows.

```
type(ID1, CPU-586) ∧ type (ID2, motherboard-2) ∧
                 conn(ID1, motherboard-port, ID2, cpu-port)    false.
```

After having translated the conceptual configuration model into the component port representation discussed in this section, the resulting knowledge base must be validated. How to support the validation process using model-based diagnosis will be discussed in the next section.

[3] A detailed discussion on the translation rules is given in (Felfernig, Friedrich, and Jannach 1999). The translation of OCL constraints is discussed in (Felfernig, Friedrich, and Jannach 2000).

4. Diagnosing the knowledge base and requirements

4.1 VALIDATING THE KNOWLEDGE BASE

In order to validate the knowledge base generated from the UML configuration model, the domain expert provides positive and negative examples (see Figure 5). We denote the set of positive examples as E^+, the set of negative examples as E^-, where $e^+ \in E^+$ and $e^- \in E^-$. 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[4] 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 restrictivness 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.

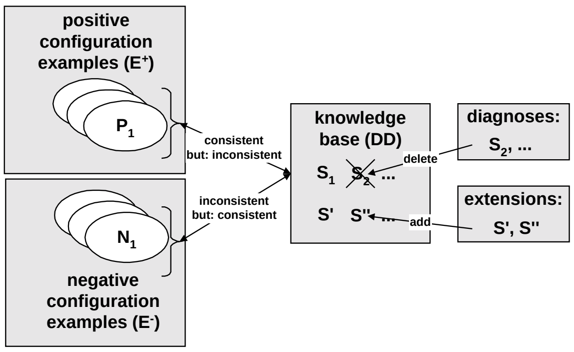
Typically, the examples will of course consist mostly of sets of *func*, *type*, *conn*, and *val* literals. In the case these examples are complete, special completeness axioms must be added.

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^-.

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

[4] Note that a configuration can contain components as well as related functions (see the example configuration result in Section 3).

and E^{-} of negative examples. The examples are given as sets of logical sentences. We assume that each example on its own is consistent. ❑



positive configuration examples (E+)
P1
consistent but: inconsistent
inconsistent but: consistent
N1
negative configuration examples (E-)
knowledge base (DD)
S1 S2 ...
S' S'' ...
delete
add
diagnoses:
S2, ...
extensions:
S', S''

Figure 5: diagnosis of configuration knowledge base

Definition (CKB Diagnosis): A CKB *diagnosis* 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 $DD - S \cup EX \cup e^{+}$ is consistent $\forall\ e^{+} \in E^{+}$ and $DD - S \cup EX \cup e^{-}$ inconsistent $\forall\ e^{-} \in E^{-}$ ❑

Additionally, we can separately diagnose the component model and the functional architectures of the configuration model, i.e. if we want to detect faulty parts of the product structure, (E^{+},E^{-}) represent positive/negative examples describing the product structure (*type*, *val*, and *conn* literals). If we want to diagnose functional architectures, (E^{+},E^{-}) represent positive/negative examples describing functional requirements (*func*, *val*, and *conn* literals). In practice such a separate diagnosis is needed, since the product structure design and the design of functional architectures are done by different people (e.g. technical experts, marketing experts etc.). Finally, we combine functional architectures and the component model in order to test whether a certain set of functional requirements can be realized by the current product model. In this case (E^{+}, E^{-}) represent functional requirements which are tested against the component model together with the functional architectures.

A diagnosis will always exist under the (reasonable) condition that the positive and negative examples do not interfere with each other.

In order to employ the hitting-set algorithm proposed by (Reiter 1987) we need to define conflict sets, which serve as input for the calculation of diagnoses[5].

[5] A detailed discussion on the algorithm for calculating CKB diagnoses is given in (Felfernig, Friedrich, Jannach, and Stumptner 2000).

Definition (CKB 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[6]. We say that, if $e^+ \in E^+$: $CS \cup e^+ \cup NE$ is inconsistent, that e^+ *induces CS*. ❑

4.2 DIAGNOSING (CUSTOMER) REQUIREMENTS

Instead of an engineer testing an altered (extended or updated) knowledge base, we are now dealing with an end user (customer or sales rep) 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 6). The domain description is used in the role of an all-encompassing partial example for correct configurations.

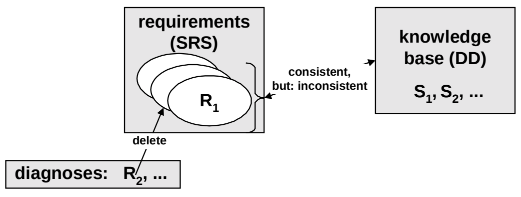


Figure 6: Diagnosing (customer) requirements

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 $SRS\text{-}S \cup DD$ 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. ❑

[6] NE denotes the conjunction of negated negative examples, i.e. $\bigwedge_{e^- \in E^-} (\neg\, e^-)$.

5. Reconfiguration

There is an increasingly demand for software supporting after-sales activities in various application domains (Männistö 1999). 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. When comparing reconfiguration with configuration, the main difference lies in an existing product which mainly influences the process of reconfiguration. The goal of reconfiguration is the identification of those parts of the existing configuration *(CONF)* which are not consistent with the domain description *(DD)* and the new (customer) requirements *(SRS')* (see Figure 7). Note that for purposes of reconfiguration the original configuration *(CONF)* is represented as a set of logical sentences representing constraints resulting in one single configuration, i.e. the facts constituting an old configuration (Figure 7) are represented as constraints.

An example for a reconfiguration task is the upgrade of an existing PC configuration *(CONF),* which must be changed in order to be consistent with the new requirements, e.g. an *SCSI disk* should be added which additionally requires an *SCSI controller*. Other examples are the reconfiguration of railway stations in order to provide more capacity, reconfiguration of telephone switching systems (extend current capacity, or change the provided functionality), or the reconfiguration of insurance policies in order to consistently adapt the policy.

Our approach does not require the explicit formulation of repair actions as discussed in (Crow, Rushby, 1991). Reconfigurations are calculated by considering constraints identified as too restrictive, i.e. do not allow the calculation of a reconfiguration based on the old configuration. For example, if the customer wants to replace the currently installed *CPU-486* with a *CPU-586*, the constraint *'CPU-486 must be part of the configuration'* is negated and added as additional constraint to the reconfiguration task.

Definition (Configuration Diagnosis): A configuration diagnosis is a set $H \subseteq CONF$, such that $CONF\text{-}H \cup DD \cup SRS'$ is consistent. ❑

Definition (Reconfiguration Problem): A reconfiguration problem is a triple (*CONF*, *SRS'*, *DD*), where *CONF* represents the old configuration, *SRS'* represents the new requirements and *DD* is the configuration domain description. A reconfiguration of a configuration *CONF* is a configuration *CONF'*, s.t. $CONF' \cup SRS' \cup DD \cup (CONF\text{-}H)$ is consistent. ❑

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

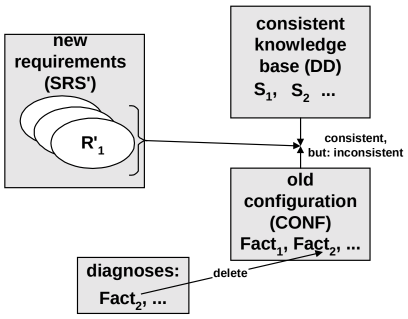


Figure 7: Diagnosing old configurations

The result of a reconfiguration step is a set of possible configurations which must be ordered conforming to some evaluation criteria. *Algorithm 1* is a simple evaluation algorithm for ordering the resulting reconfigurations *r (r* $\in$ *RCONFS*, where *RCONFS* denotes the set of calculated reconfigurations from a configuration diagnosis *x* $\in$ *H). H* denotes the set of calculated reconfiguration diagnoses for *(CONF, SRS', DD)*, *CONF'* contains the found optimal reconfiguration w.r.t. the defined *cost* function.

The structure of the *cost* function depends on the application domain. In our simple PC example we can employ a rather simple cost function which calculates the price of additionally added components. Other simple heuristics are the minimum set of exchanged components, or the minimum reconstruction effort. Further aspects which must be considered in order to determine adequate reconfigurations, are the reuse of components, the efforts for building in and removal of components, or current utilization of resources. In order to consider these additional aspects scheduling/simulation functionality could be embedded into the reconfiguration solution. This is outside the scope of this paper.

Algorithm 1: (Evaluate Reconfigurations):

```
H=diag(CONF, SRS', DD)
mincost=maxcost;
for each x ∈ H do
   RCONFS=config(CONF-x, SRS', DD)
   for each r ∈ RCONFS do
        if(cost(r, CONF) < mincost)
             mincost=cost(r);
             CONF'=r;
       endif;
   endfor;
endfor;
```

6. 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 (Lowry, Philpot, Pressburger, and Underwood 1994) is automated software reuse, where program construction is realized reusing existing software libraries. In (Bourdeau and Cheng 1995) a formal semantics for object model diagrams based on OMT (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991) 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. (Robbins, Medvidovic, Redmiles, and Rosenblum 1998) propose an integration of two ADLs (*C2* and *Wright*) 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 (Chandrasekaran, Goel, and Iwasaki 1993), 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 1989) 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 (Stumptner 1997). 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 are used in quite different application areas, e.g. diagnosis of hardware designs (Friedrich, Stumptner, and

Wotawa, 1996), diagnosis of constraint violations in databases (Gertz and Lipeck, 1995), or diagnosis of logic programs using expected and unexpected query results to identify incorrect clauses (Console, Friedrich, and Dupré, 1993). This approach differs from what we did in the sense that it uses queries and horn-clause representation in comparison with the consistency-based approach using general clauses presented in this paper.

(Crow and Rushby 1991) discuss the application of model-based diagnosis for solving reconfiguration tasks. They formulate a reconfiguration problem as a diagnosis problem, where abnormal components in the sense of model-based diagnosis are represented through components, which must be reconfigured. They propose a *rcfg* predicate in order to indicate system components which must be reconfigured. An explicit formulation of reconfiguration knowledge is needed, whereas in our approach reconfiguration knowledge is contained in the domain description. Parts which must be reconfigured are represented as negated constraints describing parts of the initial configuration setting. (Männistö, Soininen, Tiihonen, and Sulonen 1999) propose a reconfiguration approach, where reconfiguration knowledge is explicitly represented through reconfiguration operations, where an optimal reconfiguration can be calculated by evaluating the generated reconfiguration sequence. (Stumptner and Wotawa 1999) propose an approach for reconfiguring telephone networks. Reconfigurations represent adapted parameter settings consistent with the desired functions. Reconfiguration in our environment does not necessarily mean changing component parameters in the actual configuration, but rather the exchange of components and potentially the expansion of the actual configuration.

7. Conclusions

In this paper we have shown how to commonly 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 designing configuration knowledge bases on a conceptual level - 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. Further work in this area will include the automatic generation of configuration user interfaces from UML models designed using corresponding profiles.

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 (Friedrich and Stumptner, 1999), model-based diagnosis is a highly suitable technique to aid in the debugging of configurators. The proposed definition enables us to clearly identify the causes (diagnoses) that explain a misbehavior of the configurator and the unfeasibility of (customer) requirements. Furthermore, those parts of an old configuration can be identified, which must be removed from the old configuration in order to calculate a new configuration consistent with the new (customer) requirements.

The concepts presented in this paper are currently implemented in a prototype configuration environment using the ILOG Solver and Configurator libraries (C++ libraries). The configuration models are designed using the CASE tool Rational Rose. The resulting models are automatically translated into the constraint-based representation of ILOG. The basic units for diagnosing the knowledge base as well as for diagnosing (customer) requirements or old configurations are constraints imposed on a generic product structure. Our next goals are the integration of scheduling and simulation functionality in order to provide a basis for the definition of advanced ordering criteria on calculated reconfigurations.

References

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

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

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

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*, pages 1494–1499, Chambery, August 1993. Morgan Kaufmann.

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, United Kingdom, July, 1997.

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

Felfernig, A., Friedrich, G., Jannach, D., and Stumptner, M.: Consistency-based diagnosis of configuration knowledge bases, to appear: In Proceedings *European Conference on Artificial Intelligence*, 2000.

Felfernig, A., Friedrich, G., and Jannach , D., UML as domain specific language for the construction of knowledge-based configuration systems, 11th International Conference on Software Engineering and Knowledge Engineering, 1999, pp. 337-345, to appear: International Journal of Software Engineering and Knowledge Engineering (IJSEKE).

Felfernig, A., Friedrich, G., and Jannach , D., Generating product configuration knowledge bases from precise domain extended UML models, to appear: 12th International Conference on Software Engineering and Knowledge Engineering, 2000.

Fleischanderl G., Friedrich, G., Haselböck, A., Schreiner, H., and Stumptner, M.: Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems*, Vol. 13, No. 4, July/August 1998, pp. 59–68.

Friedrich, G. and Stumptner, M.: Consistency-Based Configuration, AAAI Workshop on Configuration, Technical Report WS-99-05, Orlando, Florida, 1999.

Friedrich, G., Stumptner, M., and Wotawa, F.: Model-Based Diagnosis of Hardware Designs, proc. of ECAI 1996, pages 491-496.

Gertz, M. and Lipeck, U.W.: A Diagnostic Approach to Repairing Constraint Violations in Databases. In Proceedings *DX'95 Workshop*, Goslar, October 1995.

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

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

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

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*, pages 189-196. Quality Marketing Services, Sandhurst, UK, 1998.

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

Reiter, R.: A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

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

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

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

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

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

Stumptner, M., Wotawa, F.: Reconfiguration using Model-based Diagnosis, *Proceedings of the International Workshop on Diagnosis (DX'99)*, 1999.