AN INTEGRATED DEVELOPMENT ENVIRONMENT FOR THE DESIGN AND MAINTENANCE OF LARGE CONFIGURATION KNOWLEDGE BASES

A. FELFERNIG, G. FRIEDRICH, D. JANNACH
Institut für Wirtschaftsinformatik und Anwendungssysteme
Universität Klagenfurt, 9020 Klagenfurt, Austria

AND

M. STUMPTNER
Abteilung für Datenbanken und Artificial Intelligence
Technische Universiät Wien, 1040 Wien, Austria

Abstract. Configuration problems are a thriving application area for declarative knowledge representation that experiences a constant increase in size and complexity of knowledge bases. A key issue in this context is the integrated support of configuration knowledge base development and maintenance. This paper presents an integrated development environment consisting of three major components, namely a design component which supports the conceptual design of the configuration model in UML (Unified Modeling Language) and automatic translation of the design model into a logic representation, a configuration component which allows the interactive design of concrete configurations, and finally a diagnosis component which supports the validation of the generated knowledge bases and the diagnosis of user requirements. An example for configuring computer systems shows the whole process from the design of the configuration model to the interactive configuration of the final product.

1. Introduction

Knowledge-based configuration systems have a long history as a successful AI application area and today form the foundation for a thriving industry. These systems have likewise progressed from their successful rule-based origins (Barker and O'Connor, 1989) to the use of higher level representations such as various forms of constraint satisfaction (Stumptner, Haselböck, and Friedrich, 1998), description logics (McGuinness and

Wright, 1998), or functional reasoning (Runkel, Balkany, and Birmingham, 1994), due to the significant advantages offered: more concise representation, higher maintainability, and more flexible reasoning.

The increased use of knowledge-based configurators in various application domains (e.g. telecommunication systems, automotive industry, computer systems etc.) as well as the increasingly complex tasks tackled by such systems ultimately lead to both the knowledge bases and the resulting configurations becoming larger and more complex. As a result, the user of a configuration tool, whether an engineer working on maintaining the knowledge base, or an end user producing actual configurations, is increasingly challenged, when the configuration system does not behave as expected, to find out what is actually wrong.

Apart from efficient reasoning on large and complex knowledge bases, effective knowledge acquisition is crucial since configurator development time is strictly limited, i.e. the product and the corresponding configuration system have to be developed in parallel. Additionally, the complexity of the configuration task requires sophisticated knowledge of technical experts, which must be adequately transmitted into a logical representation. For the development of product configuration systems, we show how these requirements can be met by using a standard design language (UML-Unified Modeling Language (Rumbaugh, Jacobson, and Booch, 1998)) as notation in order to simplify the construction of a logic-based description of the domain knowledge. UML is an object oriented modeling language that is widely applied in industrial software development processes. This notation is similar to approaches like entity relationship diagrams and OMT diagrams (Object Modeling Technique) and is easy to understand and communicate between domain experts.

In this paper we present an integrated development environment which includes knowledge acquisition, i.e. the construction of conceptual product models using UML, the debugging of the new/changed knowledge bases using model-based diagnosis, and finally configuration by a corresponding configuration engine.

The system architecture is shown in Figure 1. *Knowledge acquisition* is done by using specific modeling concepts of the configuration domain defined through UML stereotypes (Felfernig, Friedrich, and Jannach, 1999). The configuration models are automatically translated into a logical representation executable by a configuration engine. The *configuration environment* is the second major component. Using validated knowledge bases it supports the end-user in constructing concrete configurations. The *diagnosis component* supports both the knowledge acquisition component for validating the knowledge base, and the configuration component for validating the user requirements.

The contributions of our paper are the following. Rapid application development is facilitated through effective knowledge acquisition using configuration domain specific modeling concepts which are automatically translated into an executable logic representation. Validation and maintenance tasks are enhanced through an understandable conceptual representation of the configuration knowledge (UML model) as well as through a debugging facility (diagnosis component) for the generated configuration knowledge base. The diagnosis component is also used for detecting contradictory user requirements during the actual configuration process. Finally, all these concepts are integrated into a common development environment for knowledge-based configuration systems.

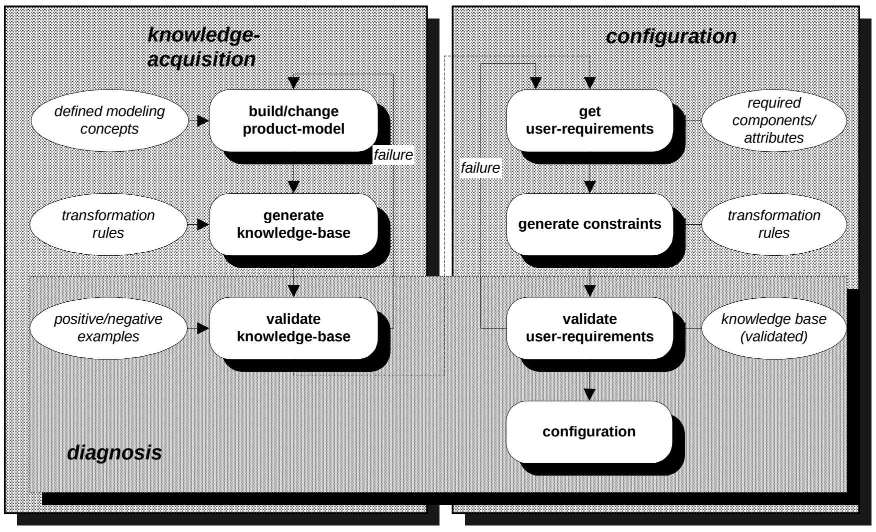

Figure 1: configuration system development environment

The paper is organized as follows. We show how the three main parts of the development environment are integrated by giving an example from the computer configuration domain. First, given a conceptual design of a configurable computer in UML, we show how this model can automatically be translated to the logical level by defining the appropriate transformation rules (Section 2). The next step is to diagnose the configuration knowledge base in order to support the validation of the knowledge base (Section 3). Third, we show how diagnosing the user requirements can support the configuration task (Section 4). In Section 5 we discuss the structure of our prototype system. Section 6 and 7 contain related work and conclusions.

2. Designing configuration models using UML

In the following we give an overview of the modeling concepts used for designing highly variant products and propose translation rules to a logical representation formalism. The applicability of these concepts for configuration problems has been shown in (Peltonen, Männistö, Alho, and Sulonen, 1994). Additionally, positive application tests were conducted in the telecommunication domain.

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

We employ the extension mechanism of UML (stereotypes) to express domain-specific modeling concepts, which has shown to be a promising approach in other areas (Robbins, Medvidovic, Redmiles, and Rosenblum, 1998). The semantics of the different modeling concepts are formally defined by the mapping of the notation to logical sentences.

In this section we give an example for a product model designed with UML stereotypes. In addition we give a formal definition of the configuration problem as logical basis for the formulation of the translation rules from the conceptual UML-level to the logical level.

2.1. USED MODELING CONCEPTS

The following example (Figure 2) shows how a configurable product can be modeled using an UML static structure diagram. This diagram describes the generic product structure, i.e. all possible variants of the product. The set of possible products is restricted through a set of constraints which relate to customer requirements, technical restrictions, economic factors, and restrictions according to the production process. Note that some component types are not refined further (*motherboard-2*, *motherboard-3*, *ide-unit*). Constraints relevant for discussing the diagnosis of the knowledge base are denoted with their identifier in the knowledge base (C*n*).

For presentation purposes we introduce a simplified model of a configurable PC as working example. We use standard UML-concepts as well as newly introduced domain-specific stereotypes, whereby their usage is restricted through OCL constraints (Object Constraint Language) in the UML metamodel. The basic structure of the product is modeled using classes, generalization, and aggregation of the well-defined parts (component types) the final product can consist of.

In the following we give a short description of the modeling concepts used for the construction of a configuration model.

Requires and incompatible ‘X *requires* Y’ means, if X is part of the product then Y must be part of the product too. ‘X *incompatible* Y’ means, if X (Y) is part of the product then Y (X) must not be part of the product.

Ports and connections For some configuration domains, not only the quantity and kind of the employed components are important but also how different components are connected to each other. Components can be connected through connection points (ports). One port can only be connected to exactly one other port.

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

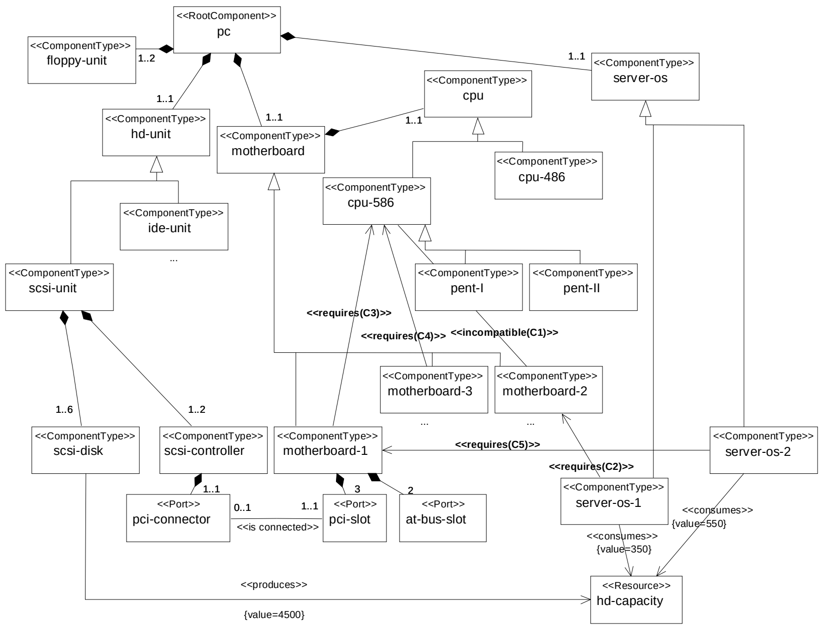
<<RootComponent>> pc
<<ComponentType>> floppy-unit
1..2
<<ComponentType>> server-os
1..1
<<ComponentType>> cpu
1..1
<<ComponentType>> hd-unit
1..1
<<ComponentType>> motherboard
1..1
<<ComponentType>> cpu-486
<<ComponentType>> cpu-586
<<ComponentType>> ide-unit
...
<<ComponentType>> scsi-unit
<<ComponentType>> pent-I
<<ComponentType>> pent-II
<<requires(C3)>>
<<requires(C4)>>
<<incompatible(C1)>>
<<ComponentType>> motherboard-3
<<ComponentType>> motherboard-2
...
...
1..6
1..2
<<ComponentType>> scsi-disk
<<ComponentType>> scsi-controller
<<ComponentType>> motherboard-1
<<requires(C5)>>
<<ComponentType>> server-os-2
<<requires(C2)>>
1..1
3
2
<<Port>> pci-connector
0..1
1..1
<<is connected>>
<<Port>> pci-slot
<<Port>> at-bus-slot
<<ComponentType>> server-os-1
<<consumes>> {value=550}
<<consumes>> {value=350}
<<produces>>
{value=4500}
<<Resource>> hd-capacity

Figure 2 : product model of a configurable PC

Resources A further enhancement of the model is expressed through resources which impose additional constraints on the possible product structure. Some components can contribute to a resource whereas others are consuming from the resource. In an actual configuration the resources must be balanced, i.e. the consumed resources must not exceed the provided resources. The contribution and consumption of a resource is modeled through relations *consumes* and *produces*. A tagged value denotes the actual value of production and consumption. In our example, the disk capacity (*hd-capacity*) of the system must be greater or equal to the capacity consumed by the installed software.

Additional modeling concepts and constraints 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 the following two steps have to be taken: First, define the new concept (a new stereotype) and state the well-formedness rules (in OCL) for its correct use within the model. Second, define the semantics of the concept for the configuration domain by stating the facts and constraints induced to the logic theory when using the concept.

If there are additional constraints in the product, which can not be expressed through predefined or newly introduced graphical concepts, one can define constraints in OCL (Object Constraint Language). As an integral part of UML, OCL allows the definition of constraints which can not be formulated using graphical concepts. As it is done for the graphical modeling concepts, OCL expressions are translated into a logical representation executable by the configuration engine.

The constraint 'if *server-os-2* is chosen then two *floppy-units* must exist in the configuration' can be expressed through the following OCL constraint:

```
self.server-os.isOclTypeOf(server-os-2) implies self.floppy-unit.size=2.
```

2.2. DEFINITION OF A CONFIGURATION PROBLEM

The following definition of a configuration problem is based on a consistency-based approach. A configuration problem can be seen as a logic theory that describes a component library, a set of constraints, and customer requirements. Components are described by attributes and ports. Ports are used as connection points between components (Friedrich and Stumptner, 1999).

The result of solving a configuration task is a set of components, their attribute values, and connections that satisfy the logic theory. This model has proven to be simple and powerful to describe general configuration problems and serves as a basis for configuration systems as well as for representing technical systems in general (Mittal and Frayman, 1989; Stumptner, 1997; Stumptner, Haselböck, and Friedrich, 1994). The model will now be treated more formally.

The formulation of a *configuration problem* can be based on two sets of logic sentences, namely *DD* (domain description) and *SRS* (specific requirements). We restrict the form of the logical sentences to a subset of range restricted first-order-logic with a set extension and interpreted function symbols. In order to assure decidability, we restrict the term-depth to a fixed number. Additionally, domain-specific axioms for configuration are defined, e.g. one port can only be connected to exactly one other port.
DD includes the description of the different component types (*types*), named ports (*ports*), and attributes (*attributes*) with their domains (*dom*).

An example from the PC configuration:

```
types={pc,cpu,cpu-586,pent-I,pent-II, cpu-486,motherboard,...}.
attributes(server-os-1)={version}.
dom(server-os-1,version)={1.0,2.0}.
ports(pc)={hd-unit-port, motherboard-port,... }.
ports(motherboard)={pc-port,cpu-port,...}.
```

DD also includes constraints to describe potential correct combinations of components, connections and value instantiations. ***SRS*** includes the user requirements on the product which should be configured. These user requirements are the input for the configuration task.

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

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 a component identifier.

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

ATTRS is a set of literals of the form *val(c,a,v)*, where *c* is a component identification, *a* is an attribute of that component, and *v* is the actual value of the attribute (selected out of the domain of the attribute).

Example for a configuration result:

```
type(p1,pc). type(m1,motherboard-1). type(c1,pent-I).
conn(p1,motherboard-port,m1,pc-port). conn(c1,motherboard-port,m1,cpu-port).
```

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

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

Definition (Consistent Configuration): Let (*DD*, *SRS*) be a configuration problem and *COMPS*, *CONNS*, and *ATTRS* represent a configuration. A configuration is consistent *iff* $DD \cup SRS \cup COMPS \cup CONNS \cup ATTRS$ can be satisfied.

Additionally we have to specify that *COMPS* includes all required components, *CONNS* describes all required connections, and *ATTRS* includes a complete value assignment to all variables in order to achieve a *complete* configuration.

This is accomplished by additional logical sentences which can be generated using the domain description. A configuration which is consistent and complete w.r.t. the domain description and the customer requirements, is called a *valid configuration*. A detailed formal exposition is given in (Friedrich and Stumptner, 1999).

2.3. TRANSFORMATION RULES

In order to allow automatic construction of the knowledge base from the conceptual model, we have to clearly define the semantics of the employed concepts. In our approach, we define the semantics through logical sentences[1] which restrict the set of possible configurations, i.e. instance models which strictly correspond to the class diagram defining the product structure. The result of the transformation is a set of first-order logical sentences that form a domain description that can be interpreted by the configuration engine.

We do not explain the complete set of transformation rules in this paper. The complete definition of the transformation rules for the modeling concepts can be found in (Felfernig, Friedrich, and Jannach, 1999). In the following we describe the rules for the derivation of *component types*, *requires*-, and *incompatible*-constraints which are further used when the configuration model is diagnosed.

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

Definition: Given a component type c in the graphical representation (*GREP*) then $c \in types$. Given an attribute a of component type c in *GREP* then $a \in attributes(c)$. Given a domain description d of an attribute a of component type c in *GREP* then $dom(c,a) = d$.

[1] We employ a logic programming notation where variable names start with an upper case letter or are written as "_". The variables are all-quantified if not explicitly mentioned. We use the unique name assumption except for skolem constants.

Requires A relation *a requires b* in *GREP* denotes that the existence of an instance of component type *a* requires that an instance of *b* exists and is part of (connected to) the same (sub-)configuration. In our example, the fact that *server-os-1* requires a *motherboard-2* implies that *server-os-1* and *motherboard-2* are part of the same *pc* which is the common and unique root of both component types in the part-of hierarchy. For the translation of the part-of relationship we use *conn*-predicates already mentioned in Section 2.2, i.e. if a *is part-of* b then a and b are connected via ports.

For the derivation of constraints expressing the *requires* and *incompatible* relation we use the abbreviations (similar to macros) *navigation_expr* and *generating_expr*, which represent path-expressions between a component and an instance of the common root (defined through *conn*-predicates). In the following example the common root of *server-os-1* and *motherboard-2* is *pc* (represented by *P*). In the case of *generating_expr* the variables are existentially quantified and the expression may only be used on the right-hand-side of the implications. The following example will illustrate this.

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

```
type(ID1,a) ∧ navigation_expr(ID1, P)
   ∃ (ID2) type(ID2,b) ∧ generating_expr(ID2, P).
```

Example: Using this generic formula the following logic expression for the *requires* relation between *server-os-1* and *motherboard-2* is derived.

```
type(ID1, server-os-1) ∧ type(P, pc) ∧ conn(ID1, pc-port, P, server-os-port)
   ∃ (ID2) type(ID2, motherboard-2) ∧ conn(ID2, pc-port, P, motherboard-port).
```

The left-hand side of the implication describes a path to the common root (*pc*). The right-hand side of the implication requires the existence and connection of the components on the path from *b* to the common root.
We defined the semantics of a *requires* relation to be "not exclusive" and of multiplicity "1..1". If more than one component requires a component of type *b*, only one instance of *b* is needed.

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

Definition: Given the relation *a incompatible b* in *GREP* where *a* and *b* are component types we extend the domain definition by the following formula:

```
type(ID1,a) ∧ navigation_expr(ID1, P) ∧ type(ID2,b) ∧ navigation_expr(ID2, P)
      false.
```

Example: Using this generic formula the following logic expression for the *incompatible* relation between *motherboard-2* and *cpu-586* is derived.
Note that the common root of *motherboard-2* and *cpu-586* is *motherboard-2*.

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

Resulting example knowledge base There are additional constraints which guarantee certain application independent properties of a configuration, e.g. connections are symmetric, a port can only be connected to one other port, or components have a unique type. These constraints are listed in the following knowledge base since they are relevant for further discussions dealing with the diagnosis of the knowledge base. However, we do not translate all component types and constraints contained in the GREP of Figure 2 because of space limitations. In the following we assume that only one instance of the root component is allowed in a configuration.

```
/* component-types, attributes and domains of the configuration-model */
types = {pc, hd-unit, ide-unit, scsi-unit, motherboard, motherboard-1,
         motherboard-2, motherboard-3, server-os, server-os-1, server-os-2, cpu,
         cpu-486, cpu-586, pent-I, pent-II...}.

/* attributes and their domains are omitted here */

/* relevant ports of the configuration-model */
ports (pc) = {floppy-unit-port1, floppy-unit-port2, hd-unit-port,
              motherboard-port, server-os-port}.
ports (floppy-unit) = {pc-port}. ports (motherboard) = {pc-port, cpu-port, ...}.
ports (server-os) = {pc-port}. ...

/* Constraint C1:  motherboard-2 incompatible cpu-586 */
type(ID1, motherboard-2) ∧ type(ID2,cpu-586) ∧
   conn(ID2, motherboard-port, ID1, cpu-port)    false.

/* Constraint C2: server-os-1 requires motherboard-2 */
type(ID1, server-os-1) ∧ type(P, pc) ∧ conn(ID1, pc-port, P, server-os-port)
   ∃ (ID2) type(ID2, motherboard-2) ∧ conn(ID2, pc-port, P, motherboard-port).

/* Constraint C3: motherboard-1 requires cpu-586 */
type(ID1, motherboard-1)
   ∃ (ID2) type(ID2, cpu-586) ∧ conn(ID2, motherboard-port, ID1, cpu-port).

/* Constraint C4: motherboard-3 requires cpu-586 */
type(ID1, motherboard-3)
   ∃ (ID2) type(ID2, cpu-586) ∧ conn(ID2, motherboard-port, ID1, cpu-port).

/* Constraint C5: server-os-2 requires motherboard-1 */
type(ID1, server-os-2) ∧ type(P, pc) ∧ conn(ID1, pc-port, P, server-os-port)
   ∃ (ID2) type(ID2, motherboard-1) ∧ conn(ID2, pc-port, P, motherboard-port).

/* Constraint C6 (application independent): connections are symmetric */
conn(ID1, P1, ID2, P2)    conn(ID2, P2, ID1, P1).

/* Constraint C7 (application independent): a port can only be
   connected to one other port */
conn(ID1, P1, ID2, P2) ∧ conn(ID1, P1, ID3, P3)    P2=P3 ∧ ID2=ID3.

/* Constraint C8 (application independent): components have a unique type */
type (ID1, T1) ∧ type (ID1, T2)    T1=T2.
```

3. Validating the configuration knowledge base

In this section we will discuss the validation of the knowledge base which was generated through application of the transformation rules described in section 2.3. Let us assume that the model in Figure 2 is the result of faulty maintenance activities which will be described in the following.

First, *motherboard-3* was introduced as new component type. What was not considered while changing the model is the fact that *server-os-1* either requires a *motherboard-2* or a *motherboard-3* but excludes the usage of a *motherboard-1*, i.e. constraint C2 is incorrect, it is too restrictive. C5 is incorrect too since a *server-os-2* can also be used in conjunction with a *motherboard-3*. Additionally, *pent-I* and *pent-II* were introduced as different processor types of type *cpu-586*. *motherboard-1* requires a *pent-I* processor which is not considered in the configuration model.

In the following we give an example how to apply model-based diagnosis (Reiter, 1987) in order to detect those constraints of the configuration model which must be changed or eliminated for achieving a valid configuration knowledge base. We denote the faulty knowledge base by KB_{faulty}, the application-independent constraints {C6..C8} by C_{Conn}.

3.1. DIAGNOSIS OF THE CONFIGURATION KNOWLEDGE BASE

In order to validate the knowledge base generated from the UML model shown in Figure 2 the domain expert provides positive and negative examples for testing the knowledge base. We denote the set of positive examples as E^+, the set of negative examples as E^-, where $e^+ \in E^+$ and $e^- \in E^-$. Note that examples can be partial configurations (e.g. some components, connections, or attributes are missing) and complete configurations as well. In the following we give two positive and one negative partial example.

First, having the correct configuration model in mind, the domain expert provides positive examples. In our application the first example is a *pc* which consists of a Motherboard of type *motherboard-3* and a Server-OS of type *server-os-1*.

e_1^+={∃ (PC, MB, SOS): type(PC, pc). type(MB, motherboard-3).
type(SOS,server-os-1). conn(PC, motherboard-port, MB, pc-port).
conn(PC, server-os-port, SOS, pc-port).}

The second example is a *pc* which consists of a *server-os-2*, a *motherboard-3*, and a *pent-II*.

e_2^+={∃ (PC, MB, CPU, SOS): type(PC, pc). type(CPU, pent-II). type(MB,
motherboard-3). type(SOS,server-os-2). conn(PC, motherboard-port, MB,
pc-port). conn(MB, cpu-port, CPU, motherboard-port). conn(PC, server-os-port,
SOS, pc-port).}

The negative example e_1^- is represented by a *pc* consisting of a *pent-II* processor and a *motherboard-1*.

```
e1-={∃ (PC, MB, CPU): type(PC, pc). type(MB, motherboard-1). type(CPU, pent-II).
   conn(PC, motherboard-port, MB, pc-port). conn(MB, cpu-port, CPU,
   motherboard-port).}
```

Testing the knowledge-base (KB_{faulty}) with e_1^+ results in a contradiction, since *server-os-1* requires a *motherboard-2,* i.e. $KB_{faulty} \cup e_1^+ \cup C_{Conn}$ is inconsistent. Testing KB_{faulty} with e_2^+ is inconsistent too. However, $KB_{faulty} \cup e_1^- \cup C_{Conn}$ is consistent what is not intended. The question is which of the application-specific constraints {C1..C5} are faulty.

As we will see the question can be answered by adopting a consistency-based diagnosis formalism (Reiter, 1987). The constraints {C1..C5} are then viewed as components and the problem can be reduced to the task of finding those constraints which, if canceled, restore consistency. The question is which set of constraints must be eliminated for making the knowledge base accepting the positive examples. In addition we have to find an extension such that the knowledge base does not accept any $e^- \in E^-$.

These concepts will be defined and generalized in the following sections. Using the consistency-based diagnosis framework will also give us the ability to identify faults given sets of multiple examples (negative and positive) and diagnose knowledge bases with multiple faults.

3.2. DEFINITION OF A DIAGNOSIS PROBLEM FOR CONFIGURATION KNOWLEDGE BASES

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. ❑

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 as currently formulated is too restrictive. Conversely, a negative example serves to check 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 from the knowledge base.

Typically, the examples will of course consist mostly of sets of *type*, *conn*, and *val* literals. In case these examples are complete configurations special completeness axioms must be added. If an example is supposed to be a complete configuration, diagnoses will not only help to analyze cases where incorrect components or connections are produced in configurations, but also cases where the knowledge base requires the generation of superfluous components or connections. The reason why it is important to give partial configurations as examples is that, if a test case can be described as a partial configuration, a drastically shorter description may suffice compared to specifying the complete example that, in larger domains, may require thousands of components to be listed with all their connections (Fleischanderl, Friedrich, Haselböck, Schreiner, and Stumptner, 1998).

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): 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 $DD - S \cup EX \cup e^-$ is inconsistent $\forall\ e^- \in E^-$ ❑

A diagnosis will always exist under the (reasonable) condition that the positive and 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* $\forall\ e^+ \in E^+: e^+ \cup \bigwedge_{e^- \in E^-} (\neg\ e^-)$ is consistent. From here on, we refer to the conjunction of all negated negative examples as *NE*, i.e. $NE = \bigwedge_{e^- \in E^-} (\neg\ e^-)$.

Proof: see (Felfernig, Friedrich, Jannach, and Stumptner, 1999).

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

3.3. COMPUTING DIAGNOSES

The above definitions allow us to employ the standard algorithms available for consistency-based diagnosis, with appropriate extensions for the domain. In particular, we use Reiter's Hitting Set algorithm (Reiter, 1987) which is based on the concept of conflict sets for focusing purposes.

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*. ❑

In the algorithm we employ a labeling that corresponds to the labeling of the original HS-DAG (Reiter, 1987; Greiner, Smith, and Wilkerson, 1989), i.e. a node *n* is labeled by a conflict *CS*(*n*), edges leading away from *n* are labeled by logical sentences $s \in CS(n)$. The set of edge labels on the path leading from the root to *n* is referred to as *H*(*n*). In addition, each node is labeled by the set of positive examples *CE*(*n*) that have been found to be consistent with $DD\text{-}H(n) \cup NE$ during the creation of the DAG. The reason for introducing the label *CE*(*n*) is the fact that any e^+ that is consistent with a particular $DD\text{-}H(n) \cup NE$ is obviously consistent with any *H*(*n*)' such that $H(n) \subseteq H(n)'$. Therefore any e^+ that has been found consistent in step 1.(a) below does not need to be checked again in any nodes below *n*.

Since we generate a DAG, a node *n* may have multiple direct predecessors (we refer to that set as *preds*(*n*) from here on), and we will have to combine the sets *CE* for all direct predecessors of *n*. The consistent examples for a set of nodes *N* (written *CE*(*N*)) are defined as union of the *CE*(*n*) for all $n \in N$.

Algorithm (schema): Input: DD, E^+, E^-; Output: a set of diagnoses *S*

1. Use the Hitting Set algorithm to generate a pruned HS-DAG *D* for the collection F of conflict sets for (DD,E^+,E^-). The DAG is generated in a breadth-first manner since we are interested in generating diagnoses in order of their cardinality.

   (a) Every theorem prover call $\text{TP}(DD–H(n), E^+ – CE(preds(n)),E^-)$ at a node *n* corresponds to a test of whether there exists an $e^+ \in E^+ – CE(preds(n))$ such that $DD–H(n) \cup e^+ \cup NE$ is inconsistent. In this case it returns a conflict set $CS \subseteq DD–H(n)$, otherwise it returns *ok*. Let $E_{CONS} \subseteq E^+ – CE(preds(n))$ be the set of all e^+ that have been found to be consistent in the call to TP.

   (b) Set $CE(n) := E_{CONS} \cup CE(preds(n))$.

2. Return $\{H(n) \mid n \text{ is a node of } D \text{ labeled by } ok\}$

Note that in the case that e^+ is a partial configuration, each call to TP (since it involves checking the consistency of $\forall e^+ \in E^+: DD–H(n) \cup e^+ \cup NE$) corresponds to the extension of e^+ to a complete configuration.

Now we apply the algorithm for consistency-based diagnosis of configuration knowledge bases for diagnosing the faulty knowledge base

(KB_{faulty}). E^+ and E^- are represented by the examples $e_{1,2}^+$ and e_1^- given in Section 3.1. Testing e_1^+ the first TP-call returns the conflict-set {C2}, i.e. $\{C2\} \cup e_1^+ \cup NE$ is inconsistent. The result of the next TP-call is {C5} as conflict-set, since it is in contradiction with e_2^+. Finally, {C2, C5} is the diagnosis – no further constraints must be eliminated.

Figure 3 shows the calculation of diagnoses for the faulty configuration model. '✓' indicates that no further TP-call in this part of the search tree is necessary – a diagnosis is found, i.e. $\forall_{i\ (i=1..2)}$ DD - H(n) $\cup\ e_i^+ \cup NE$ is consistent. Traversing the DAG of Figure 3 we get one diagnosis, namely {C2, C5}. This diagnosis is presented to the user of the design environment indicating which parts of the configuration knowledge base contain faulty constraints and must be analysed for fixing the bugs. Since each part of the knowledge base can be non-ambigously associated with corresponding parts in the conceptual model, the diagnosis {C2, C5} can be presented to the user by indicating (e.g. highlighting) those parts in the conceptual model.

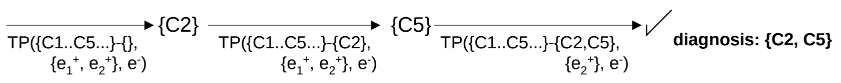
{C2}
{C5}
TP({C1..C5...}-{}, {e1+, e2+}, e-)
TP({C1..C5...}-{C2}, {e1+, e2+}, e-)
TP({C1..C5...}-{C2,C5}, {e2+}, e-)
✓
diagnosis: {C2, C5}

Figure 3 : calculating diagnoses for the example knowledge base

In our case the result is a very simple DAG. Generally there are a number of diagnoses which must be ranked. We are not going to discuss ranking criteria in this paper, for further information see (Felfernig, Friedrich, Jannach, and Stumptner, 1999).

4. Diagnosing user requirements and configuring the product

Even once the knowledge base has been tested and found correct, diagnosis can still play a significant role in the configuration process, but the scenario has changed. Instead of an engineer testing an altered (extended or updated) knowledge base, we are now dealing with an end user (e.g. customer or sales rep) who is using the tested (and assumed correct) knowledge base for configuring actual systems. During their sessions, 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. The domain description is used in the role of an all-encompassing partial example for correct configurations.

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. ❑

Remark: *S* is a CREQ diagnosis for (*SRS*,*DD*) *iff S* is a CKB diagnosis for (*SRS*,{*DD*},{}).

After having designed the configurable product and validated the generated knowledge base the actual configuration can take place. For the purposes of this example we consider the knowledge base diagnosed in Section 3.3 and assume that the faulty constraints are already eliminated or substituted by the correct ones. The user (the customer) can provide some input data and specify the requirements for the actual variant of the product.

Let the customer requirement be that the *server-os-2* has to be installed on the system together with a *cpu-486*, a *floppy-unit* and an *scsi-disk*. These requirements can be expressed as constraints, i.e.

```
/* Constraint C9: a server-os-2 must be part of the configuration */
type(P,pc)    ∃ (SOS) type(SOS, server-os-2) ∧
              conn(P,server-os-port,SOS,pc-port).

/* Constraint C10: a cpu-486 must be part of the configuration */
type(M,motherboard)   ∃ (CPU) type(CPU,cpu-486) ∧
              conn(M,cpu-port,CPU,motherboard-port).

/* Constraint C11: a floppy-unit must be part of the configuration */
type(PC,pc)   ∃ (FDU) type(FDU,floppy-unit) ∧
              conn(PC,floppy-unit-port,FDU,pc-port).

/* Constraint C12: an scsi-unit must be part of the configuration */
type(PC,pc)   ∃ (HDU) type(HDU,scsi-unit) ∧ conn(PC,hd-unit-port,HDU,pc-port).
```

The diagnosis-component detects the incompatibility between *server-os-2* and *cpu-486* since *server-os-2* requires a *motherboard-1* (or a *motherboard-3*), but *motherboard-2* is the only motherboard compatible with *cpu-486*. The diagnoses presented to the user are {C9} and {C10}. Since the constraints C9, C10, C11, and C12 can be non-ambigously associated with user-requirements, the diagnoses can be presented to the user by indicating (e.g. highlighting) the requirements causing the inconsistencies. Assuming that the user chooses the *cpu-586* (*pent-I*), the configuration system is able to generate a consistent and complete configuration.

5. Prototype environment

We have implemented a prototype development environment supporting the proposed development process (Figure 1) for configuration systems using standard commercial tools (Rational Rose, Ilog Configurator, Microsoft Visual C++). After having defined the conceptual model in Rational Rose, our prototype system generates the knowledge base for the target configuration engine (C++ code). After automated compilation and linkage, the calculation of all solutions is done by the solver software.

The diagnosis component is implemented using C++ and uses the knowledge base derived from the conceptual model. The examples can be defined in terms of partial or complete configurations. The diagnosis algorithm can be used with different search heuristics and can be restricted to a certain search depth, e.g. only single faults should be found. Diagnosing our simple example problem (Section 3) takes about one second, but the performance of the algorithm strongly depends on the number of the constraints, the employed search heuristics, and the cardinality of the diagnoses.

In addition to the example given in this paper, we have evaluated our approach on a real-world problem (private telephone switching systems). This test case showed the applicability of the conceptual modeling language as well as the effectiveness of the resulting configuration knowledge base.

6. Related work

There is a long history in developing configuration tools in knowledge-based systems (Stumptner, 1997). Progressing from rule-based systems like R1/XCON (Barker and O’Connor, 1989) higher level representation formalisms were developed, i.e. various forms of constraints satisfaction (Stumptner, Haselböck, and Friedrich, 1998), description logics (McGuinness and Wright, 1998), or functional reasoning (Runkel, Bakany, and Birmingham, 1994). (Heinrich and Jüngst, 1991) propose a resource-based paradigm of configuration where the number of components of a particular type occurring in the configuration depends on the amount of that resource required. Conforming these paradigms various configuration systems were developed (Yu and Skovgaard, 1998; Fleischanderl, Friedrich, Haselböck, Schreiner, and Stumptner, 1998; Haag, 1998).

Case based reasoning (CBR) (Kolodner, 1993) is a successful AI technique that can be employed for product configuration design (Smith and Faltings, 1994; Rahmer and Voss, 1996). Using CBR to solve configuration tasks requires cases from past configuration problems to be stored and organized suitably. Given a new task, the customer requirements are matched with the requirements of past cases. If the requirements only match partially, old solutions are fixed in an adaptation process to meet the actual requirements. However, the focus of these techniques is to find solutions to configuration problems by employing indexing and case retrieval techniques, whereas in our approach we employ cases as positive and negative examples in order to diagnose a configuration knowledge base.

The automated generation of logic-based knowledge bases through translation of domain specific modeling concepts expressed in terms of a standard design language like UML has not been discussed so far. Comparable research has been done in the fields of automated and Knowledge-based Software Engineering (Lowry, Philpot, Pressburger, and Underwood, 1994). (Bourdeau and Cheng, 1995) define a formal semantics for object model diagrams based on OMT in order to support the assessment of requirement specifications. We view our work as complementary since our goal is the generation of executable logic descriptions.

Model-based diagnosis techniques are used in quite different application areas, e.g. diagnosis of hardware designs (Friedrich, Stumptner, and Wotawa, 1999), 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), a line of work later continued by Bond (Bond, 1994; Bond, 1996). 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.

7. Conclusions

With the growing relevance and complexity of AI-based applications in the configuration area, the usefulness of other knowledge-based techniques for supporting the development and maintenance of these systems is likewise growing. We have presented a framework for an integrated environment supporting automatic generation of executable logic representations out of design descriptions, integrated debugging support for the generated knowledge bases and finally the execution of the validated knowledge bases including validation of the user requirements.

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 user requirements. Positive and negative examples, commonly used in testing configurators, are exploited to identify possible sets of faulty clauses in the knowledge base. Building on the analogy between the formal models of configuration and diagnosis, we have given an algorithm for computing diagnoses in the consistency-based diagnosis framework.

Extensible standard design languages (like UML) are able to provide a basis for introducing and applying rigorous formal descriptions of application domains. This approach helps us to combine the advantages of various areas. First, high level formal description languages reduce the development time and effort significantly because these descriptions are directly executable. Second, standard design techniques like UML are easy to apply and widely adopted in the industrial software development process.

References

Barker, V.E. and O'Connor, D.E.: 1989, Expert systems for configuration at Digital: XCON and beyond. *Comm. ACM*, 32 (3), pp. 298–318.

Bond, G.W.: 1994, Logic Programs for Consistency-Based Diagnosis. *PhD thesis*, Carleton University, Faculty of Engineering, Ottawa, Canada.

Bond, G.W.: 1996, Top-down consistency based diagnosis. In Proceedings *DX'96 Workshop*, Val Morin, Canada, pp. 18-27.

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.

Console, L., Friedrich, G., and Dupré, D.T.: 1993, Model-based diagnosis meets error diagnosis in logic programs. In Proceedings *International Joint Conference on Artificial Intelligence*, Chambery, Morgan Kaufmann, pp. 1494-1499.

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

Felfernig, A., Friedrich, G., Jannach , D., and Stumptner, M.: 1999, Consistency based diagnosis of configuration knowledge-bases. *AAAI-Workshop on Configuration*, Orlando/Florida, pp. 41-47.

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

Friedrich, G. and Stumptner, M.: 1999, Consistency-Based Configuration, *AAAI-Workshop on Configuration*, Orlando/Florida, pp. 35-40.

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

Gertz, M. and Lipeck, U.W.: 1995, A Diagnostic Approach to Repairing Constraint Violations in Databases. In Proceedings *DX'95 Workshop*, pp. 65-72.

Greiner, R., Smith, B.A., and Wilkerson, R.W.: 1989, A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88.

Haag, A.: 1998, Sales Configuration in Business Processes, *IEEE Intelligent Systems*, Vol. 13, No. 4, pp. 78–85.

Heinrich, M. and Jüngst, E.W.: 1991, A resource-based paradigm for the configuring of technical systems from modular components. In Proceedings of the 7th IEEE Conference on AI applications(CAIA), pp. 257-264.

Kolodner, J.: 1993, Case-Based Reasoning, Morgan Kaufmann.

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.

McGuinness, D.L. and Wright, J.R.: 1998, Conceptual Modelling for Configuration: A Description Logic-based Approach. A*rtificial Intelligence for Engineering Design, Analysis and Manufacturing, Special Issue: Configuration Design,* 12(4), pp. 333–344.

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

Peltonen, H., Männistö, T., Alho, K., and Sulonen, R.: 1994, Product configurations—an application for prototype object approach. *Object Oriented Programming, ECOOP'94*, Springer, pp. 513-534.

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

Rahmer, J., Voß, A.: 1996, *Case based reasoning in the Configuration of Telecooperation Systems*. AAAI'96 Fall Symposium "Configuration", AAAI Press.

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

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

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

Runkel, J.T., Balkany, and A., Birmingham, W.P.: 1994, Generating non-brittle Configuration-design Tools. In Proceedings *Artificial Intelligence in Design '94,* Lausanne, Kluwer Academic Publisher, pp. 183-200.

Smith, I., Faltings B.: 1994, *Spatial design of complex artifacts using cases*, Proc. 10th International Conference on Artificial Intelligence for Applications, IEEE, pp. 70-76.

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

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

Stumptner, M., Haselböck, A., and Friedrich, G.: 1998, Generative Constraint-Based Configuration of Large Technical Systems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Special Issue: Configuration Design,* 12(4), pp. 307–320.

Yu, B. and Skovgaard, H.J.: 1998, A configuration tool to increase product competitiveness, *IEEE Intelligent Systems*, Vol. 13, No. 4, pp. 34–41.