Knowledge Engineering for Configuration Systems

Gerhard Friedrich, Dietmar Jannach, Markus Stumptner and Markus Zanker

Abstract

Developing a product configuration system is a non-trivial and challenging task for various reasons. First, the domain knowledge which has to be encoded into the system is often spread over several departments or functions within a company. Beside that, in many cases data from existing information systems have to be integrated into the configurator. Finally, the business rules or technical constraints which define the space of possible configurations can be relatively complex and also subject to frequent changes. This makes acquiring and encoding domain knowledge as well as testing and debugging particularly demanding tasks.

In this chapter, we give an overview of the challenges when developing a knowledge based configuration system. We will particularly focus on questions related to the knowledge acquisition process and will additionally show how model-based debugging techniques can be applied to support the knowledge engineer in the testing and debugging process.

1 Introduction

Product configurators are knowledge-intense and complex software systems whose design, development and implementation within a company is a challenging and demanding undertaking throughout the whole software life cycle. Even if an off-the-shelf and generic configuration problem solving framework or a constraint engine such as the one described by (Mailharro, 1998) is used as a basis for the configurator application, there are still several aspects that contribute to the complexity of the overall development process.

Since we are only interested in non-trivial configuration systems in the context of this book, we will start from the premise that a knowledge-based approach is chosen. Thus, the configuration knowledge has to be represented by some form of logic, declarative rules or constraints. In contrast to standard software development processes, a "knowledge engineering" phase – see, for instance, (Studer et al., 1998) as an introduction to this topic – has to be part of the life cycle. In this phase, the desired configuration logic has to be modeled using the given knowledge representation mechanism of the configurator framework. The elicitation and error-free formalization of the knowledge however requires the tight co-operation between the domain expert and a knowledge engineer, where the latter needs to establish a link between the business and technologists. One of the problems in that context is that there might be several departments in a company who have an influence on the set of configurations that can be offered to a customer. Beside technical restrictions due to, for instance, component incompatibilities or logistics, also sales and marketing considerations might determine the range of allowed configurations. Overall, a large number of stakeholders and knowledge sources have to be involved into the process. Another issue in the knowledge acquisition process is the question of the choice of appropriate tools and representations mechanisms. In modern software engineering processes, conceptual modeling approaches such as OMG's Unified Modeling Language[1] or Domain-Specific Languages (DSLs) (Fowler, 2010) are common in industry today. These graphical (domain) models are not only used in the software design phase, but should also serve as a basis for the communication between the requirements engineer or system analyst and the domain experts. In Section 2, we will focus on these issues in knowledge acquisition and present a recent approach based on conceptual modeling in more detail.

Beside the noted differences in the knowledge acquisition and design phase, the declarative nature of the knowledge-based development approach also influences the required test and maintenance processes. Standard debugging approaches based on breakpoints, variable inspection, or stepping through execution traces cannot be applied to most reasoning engines which, for example, implement automatic backtracking search, rule inferencing, or constraint propagation. Even though some commercial configuration tools such as the constraint-based one described in (Junker, 2001) provided an explanation facility that goes beyond simple propagation traces, the debugging and (regression) testing support is mostly very limited in today's configuration systems. Note that this situation is particularly problematic because in many domains, the configuration knowledge bases are subject to frequent changes, such as, for instance, the telecommunication domain (cf. (Fleischanderl et al., 1998) or (Felfernig et al., 2002)). In Section 3, we will discuss a recent and generic method for debugging declarative knowledge bases, which is based on model-based diagnosis techniques. It is particular about this method that it is not limited to one single type of configuration reasoning and that it relies on partial and complete test cases in the error location process. This makes the method also naturally applicable for regression testing.

[1] UML - www.uml.org.

The final aspect is related to the development process of configuration systems and deals with the embedding of the knowledge engineering effort into the whole process of implementing a Mass Customization strategy. As already noted, configuration knowledge-bases are usually not defined from scratch by a single domain expert in collaboration with a knowledge engineer but a variety of organizational and business issues are related to this effort. The integration of the configuration process into the companies' business processes, for instance, sharing of existing components and assemblies within product family architectures as well as cost, schedule, and volume targets will involve a number of different stakeholders and experts from within an organization. Furthermore, lifetime aspects of a configuration application such as evolution of product architectures and maintenance of the configuration knowledge request attention to documentation systems and knowledge management. Consequently, we will briefly discuss these organizational aspects in Section 4 and point to related work for further study.

2 The Configurator Development Life Cycle

One of the first steps when developing a configurator application is to collect and organize the required domain knowledge. When following a knowledge-based approach, the knowledge has to be formalized based on the knowledge representation mechanism of the configuration problem solver, a phase which does not exist in standard software engineering processes.

In the knowledge acquisition process, a knowledge engineer or system analyst has to identify the different stakeholders and sources of information. Many pieces of information may however already exist in some other software system. Information about different aspects of a component may for example be available in the company's ERP system; calculations for the pricing process may be found in spreadsheet applications. This issue of embedding the configurator development into the organization's processes will be further addressed in Section 4. Regarding the planning process for the development cycle, note that in some domains, it might be necessary that the configurator application has to be developed in parallel with the configurable product in order to support integration testing (Fleischanderl et al., 1998). In these situations, the knowledge to be acquired may still be unstable and continuously changing. Consequently, the choice of the right knowledge acquisition techniques is crucial. In this section, we will therefore focus on aspects of how to acquire the configuration model in a correct and efficient way.

One of the earliest knowledge-based, large scale configurator application was DEC's R1 (XCON) rule-based "configurer" for VAX-11/780 computer systems (McDermott, 1982). Beside the fact that this was also one of the first expert systems successfully implemented in industry, it is also one of the few applications where an experience report regarding engineering and implementation issues has been published after ten years of practical use. In their report, (Barker et al., 1989) identify various challenges when developing a large-scale configurator, many of which still apply today after more than 20 years of configuration system development in industry.

From the business and strategic perspective, one main problem then and still today is to establish an appropriate link between the business and the "technologists" (the configurator developers). This aspect is particularly important since offering configurable products on the market is a strategic decision and requires the coordination of several business functions. At DEC at that time, it was also important to have a strong commitment from management to build systems with then-innovative but immature expert systems technology, which additionally required long learning and training phases for newly-hired developers. Today, declarative technology and in particular constraint-based approaches have reached some level of maturity but are still far from being "standard" tools in industrial software development practices. Even though some approaches such as the one by (Agarwal and Tanniru, 1992) toward a structured development process for the case of rule-based systems have been made, the question of the proper integration of knowledge-based system development phases into standard software engineering processes (think, e.g., of testing of such systems) is still largely open today.

From the perspective of the application domain, (Barker et al., 1989) identify the problems of volatility, scope expansion and complexity. In their domain, about 40% of the rules changed per year. Scope expansion means that the system has become more and more integrated with other tools and was used by different business groups, so that various other systems had to be connected with the configurator and additional functionality was continuously required. Regarding the complexity aspect, Barker et al. report that the rule base comprised more than 10,000 rules after ten years and that the component database contained more than 30,000 different parts.

The question of how configuration knowledge is represented has a strong influence on the knowledge acquisition and maintenance process. In the R1 system, the configuration logic was encoded by (IF-THEN-style) production rules. Today, such rule-based systems are no longer recognized as state-of-the-art in configuration systems or in knowledge-based systems development in general. Despite some approaches toward increasing the maintainability of rule-based systems, e.g., through modularity (Steve and Davis, 1990), the maintenance of such systems is considered problematic in particular due to the possible side-effects of individual rules and that the rule application strategy determines their semantic interpretation. Another problem related to the maintenance of rule-based systems is the typical mixture of different types of knowledge in one rule-base (Studer et al., 1998), where domain knowledge ("component A is incompatible with component B") is intermingled with knowledge that governs the solution search process ("try components of type A first"). In that context, note that due to the usually very large space of possible configurations, problem-solving heuristics still play an important role also in more recent approaches to building configurator applications. Such heuristics can include rules such as "fill a frame from left to right" in the telecommunication switch configuration problem (Fleischanderl et al., 1998) but also variable ordering or value selection strategies in constraint-based approaches. However, for example in constraint-based or preference-based approaches such as (Mailharro, 1998) or (Junker and Mailharro, 2003), this knowledge would be encoded differently than the compatibility constraints and does not have an impact on the set of possible configurations.

From the particular perspective of the configuration domain, another problem of rule-based systems is the limited correspondence between the structure of the knowledge base and the rule base. This missing correspondence is also one of the problematic aspects of systems that work with simple compatibility tables such as some small-scale commercial configurator solutions. Modern approaches to building configurator applications therefore commonly rely on the "component-port" model (Mittal and Frayman, 1989) in one or the other way. In that approach, a configurable system is always considered to be assembled from (configurable) components which can be connected to each other by pre-defined connection points called ports. This general model is the basis for both, constraint-based approaches such as (Mailharro, 1998) or (Fleischanderl et al., 1998) or logic-based ones like (McGuinness and Wright, 1998), and conceptual-modeling based approaches, see, e.g., (Felfernig et al., 2000b). The main advantage of the component-port model is that the modeled artifacts are often directly related to physical components, thus structuring the knowledge base and making it easier to comprehend and maintain for knowledge engineers.

2.3 The Importance of Frameworks and Tool Support

In the context of this chapter, we limit ourselves to knowledge-based approaches and at the same time assume that a pre-existing configuration framework or reasoning engine (or "expert system shell" in former years) is used. Only in rare cases such as reported in (Fleischanderl et al., 1998), companies will be able to afford the development of a proprietary reasoning engine which is tailored to the specific requirements of the domain. In the light of the many problems that can arise during knowledge engineering and maintenance, one should therefore not only pay attention to the efficiency or performance of the underlying reasoning engine but also to the availability of tools for knowledge acquisition when deciding on the technological basis for a configuration application.

A modern configuration framework should provide mechanisms which abstract from the underlying technical representation as far as possible in the modeling phase. Typical elements that can be found in such tools may for example include mechanisms such as compatibility tables, which are understandable even by non-IT people, user-oriented rule languages including appropriate editing support, or graphical representations of bill-of-material structures. A framework for such mechanisms and the tool support that can be provided will be presented in Section 2.4. In addition, appropriate support for testing, debugging, and tracing should be included in the framework in order to facilitate its practical use. We describe such a framework in Section 3.

2.4 A UML-Based Approach

Having discussed the various issues in knowledge acquisition for configurator applications, we will describe the method from (Felfernig et al., 2000b) for configurator development based on the Unified Modeling Language (UML) in this subsection. Figure 1 gives the general overview on the proposed development process.

The boxes in the figure show the different steps of the development process from product modeling to productive use. The support with appropriate mechanisms and tools for the development phases (1) to (4) is at the core of the proposal of (Felfernig et al., 2000b).

In phase (1), the knowledge engineer together with the domain expert uses the Unified Modeling Language to develop a model of the configuration problem [2].

[2] See also the discussion of viewing knowledge engineering as a modeling process vs. knowledge engineering as a transfer process in (Studer et al., 1998).

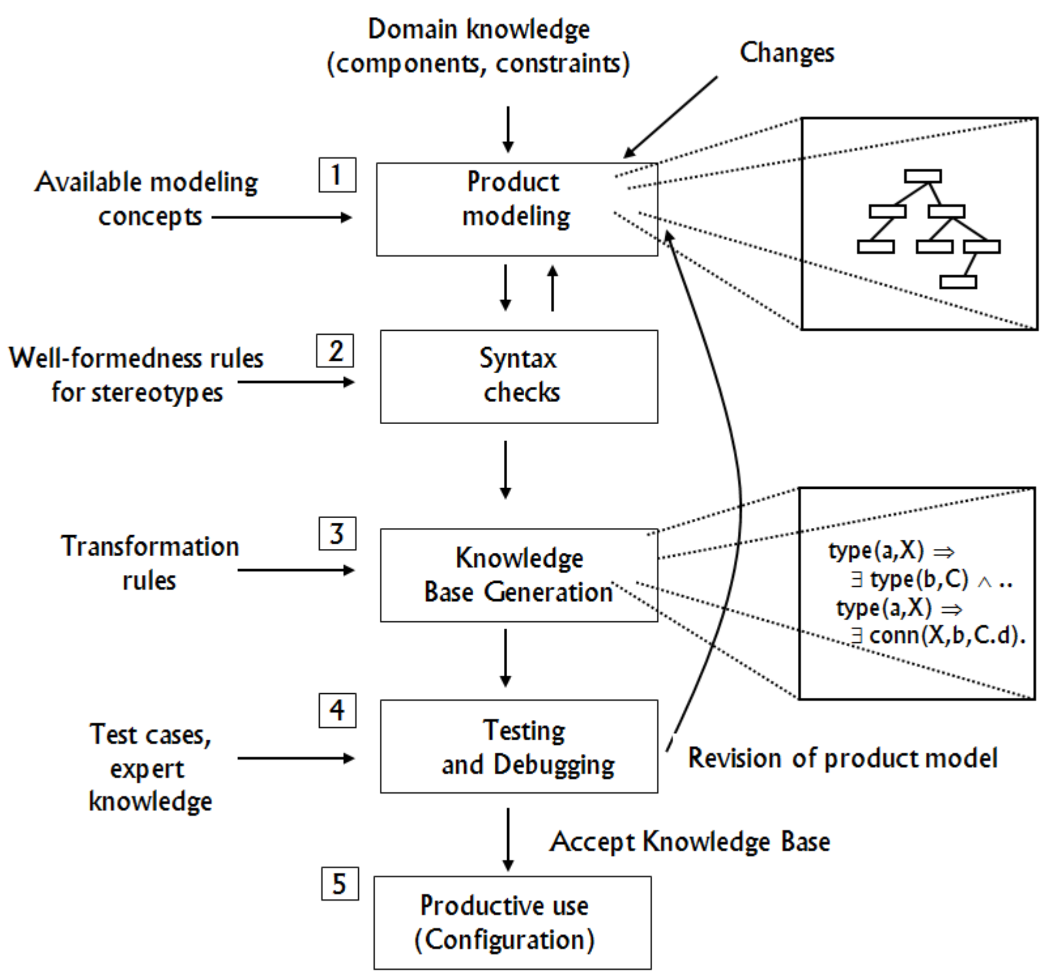
Domain knowledge
(components, constraints)
Changes
1
Available modeling concepts
Product modeling
2
Well-formedness rules for stereotypes
Syntax checks
3
Transformation rules
Knowledge Base Generation
type(a,X) ⇒
∃ type(b,C) ∧ ..
type(a,X) ⇒
∃ conn(X,b,C.d).
4
Test cases, expert knowledge
Testing and Debugging
Revision of product model
Accept Knowledge Base
5
Productive use (Configuration)

Fig. 1. Configurator development process, adapted from (Felfernig et al., 2000b).

However, instead of using some proprietary configuration-specific representation format, the idea is to rely on UML as a graphical notation, which is a widely-known and standardized language for conceptual modeling in standard software engineering processes. Since UML is a general language not tied to a specific application domain, Felfernig et al. propose to provide the modeler with a pre-defined set of configuration-specific modeling concepts. The definition of this domain-specific extensions is based on *profiles* and *stereotypes*, UML's built-in mechanism to create a domain-specific language.

In Section 2.2 we discussed the importance of appropriate problem-solving heuristics when reasoning about large-scale configuration problem instances. In many cases, this sort of fine-tuning the search process takes place at the final stages of the development process and at a low technical level based on the first experiences of running the system in production. If knowledge about effective search strategies is however already available in the earlier development stages, one could try to incorporate this knowledge also into the conceptual model. Being oriented toward the domain expert and knowledge engineer, the

UML-based approach of (Felfernig et al., 2000b) does not comprise modeling concepts for such heuristics. While there have been some proposals in the 1990s within the fields of Knowledge Acquisition and Knowledge-Based Systems which aimed at supporting the domain expert in formulating problem-solving knowledge at a more abstract or conceptual level, we see this problem to be still largely open in the configuration domain.

For each domain-specific concept, such as, for instance, the concept "component-type", a set of so-called well-formedness rules in the form of OCL (Object Constraint Language) constraints are provided, which describe how the modeling concept can be properly used. Since the proposed extensions and the OCL constraints are developed using the standard mechanisms of UML, off-the-shelf UML tools can be applied to check the syntactic validity of the configuration models in phase (2).

Subsequently, in phase (3), the conceptual models should be automatically translated into an "executable" representation, which can be interpreted by a configuration problem solver. This corresponds to the ideas of model-driven software development and automatic code generation in standard software development processes. In analogy to the Model-Driven Architecture (MDA)[3], the conceptual models can be seen as "platform independent models". The translation of the models into executable code however means that precise semantics for the given UML modeling concepts are required. In (Felfernig et al., 2000b) and (Felfernig et al., 2000a), this definition of the semantics is made by describing for each modeling construct how it translates into a general logic-based formalization of the configuration problem. In order to demonstrate the applicability of the approach also for specific realizations of the component-port approach to configuration, a mapping of the conceptual models to the constraint-based representation of the commercial ILOG Configurator framework (Mailharro, 1998) has been implemented.

In phase (4), the resulting knowledge base should be validated based on test cases and expert knowledge, before it is set into productive use in phase (5). In order to support the knowledge engineer in this phase, a debugging approach for declarative configuration knowledge bases is proposed in (Felfernig et al., 2004). The proposed method relies on model-based diagnosis techniques which were originally developed for the analysis of hardware components and electronic circuits. With the help of this method, the knowledge engineer is pointed to potentially erroneous statements within the knowledge base. Since there is however a close correspondence between the knowledge base contents and the conceptual models through the automatic generation process, the engineer can use these pointers to potential errors to fix the problems at the conceptual level and then repeat the process of knowledge base generation.

[3] http://www.omg.org/mda/

We will discuss more details of the employed model-based diagnosis technique in the next section.

In summary, one of the key goals of the proposed approach, which is based on model-based development, declarative knowledge representation, and AI-based reasoning, is to decrease the required development, debugging, and maintenance efforts. An exact scientific quantification of potential cost savings when using different software engineering techniques or a model-based approach is hard in general. An anecdotal example is given for example in (Fleischanderl et al., 1998), where the authors report that in their experience up to 20% of the product development cost in their domain usually goes into the development of the configurator software. Furthermore, they estimate that using a declarative, constraint-based approach helped them to reduce the costs for configurator development by up to 60%. A further indicator that the proposed concepts are valuable in practice is that some of them were implemented in commercial tools: The ILOG JConfigurator product of the mid-2000s, for example, included both a graphical modeling component similar to the UML-based structure diagrams from (Felfernig et al., 2000b) as well as an algorithm for conflict detection and explanation, which are only possible in a declarative approach.

3 Debugging Configuration Knowledge Bases

In order to introduce the diagnosis of configuration knowledge bases, we employ a small part of the example introduced and illustrated in Chapter **??**. All necessary axioms are presented to make the example self contained. We illustrate our approach by introducing a typical error and subsequently demonstrate the application of model-based diagnosis methods to locate the faulty description. Based on this example we describe the basic concepts of model-based diagnosis and finally show their application to our introductory example.

3.1 Motivating Example

Let us assume that besides the components which were mentioned in the introductory example, there is also a *universal motherboard* which can be part of valid PC configurations. This motherboard allows to combine a `CPUD` and a `CPUS` on one motherboard. Consequently, the configuration knowledge base **KB**[4] has to be extended by axioms[5], which reflect these new configuration options. In

[4] KB is denoted as C_{KB} in Chapter **??**.

[5] We employ the usual logic-programming notation for axioms. Symbols starting with an upper-case letter denote logical variables. All logical variables are universally

particular, a class hierarchy formed by is-a relations between various types of motherboards can be modeled as follows. Every specialized motherboard is a motherboard (Axioms 1 to 3). The specialization is exclusive (Axioms 4 to 6). Finally, Axiom 7 expresses that every motherboard must be of one specific motherboard subtype. However, in Axiom 7 the knowledge engineer forgot to extend the consequent of the formula by `MBUniversal`. We denote the faulty knowledge base by $\mathbf{KB}_{faulty}$.

(1) $\texttt{MBSilver}(X) \rightarrow \texttt{MB(X)}$.
(2) $\texttt{MBDiamond}(X) \rightarrow \texttt{MB(X)}$.
(3) $\texttt{MBUniversal}(X) \rightarrow \texttt{MB(X)}$.

(4) $\texttt{MBSilver}(X) \wedge \texttt{MBDiamond}(X) \rightarrow \texttt{false}$.
(5) $\texttt{MBSilver}(X) \wedge \texttt{MBUniversal}(X) \rightarrow \texttt{false}$.
(6) $\texttt{MBDiamond}(X) \wedge \texttt{MBUniversal}(X) \rightarrow \texttt{false}$.

(7) $\texttt{MB(X)} \rightarrow \texttt{MBSilver(X)} \vee \texttt{MBDiamond(X)}$.

The correct Axiom 7 should read:

(7^{ok}) $\texttt{MB(X)} \rightarrow \texttt{MBSilver(X)} \vee \texttt{MBDiamond(X)} \vee \texttt{MBUniversal(X)}$.

Furthermore, we declare that `CPUD` and `MBSilver` as well as `CPUS` and `MBDiamond` are incompatible, which is expressed by:

(8) $\texttt{CPUD}(C) \wedge \texttt{cpu-of-mb}(C,M) \wedge \texttt{MBSilver}(M) \rightarrow \texttt{false}$.
(9) $\texttt{CPUS}(C) \wedge \texttt{cpu-of-mb}(C,M) \wedge \texttt{MBDiamond}(M) \rightarrow \texttt{false}$.

After specifying the knowledge base the knowledge engineer will usually try to validate it with the help of test cases. When debugging configuration knowledge bases, we can distinguish between positive and negative cases. Positive cases are configurations which must be consistent with **KB** whereas negative cases have to be inconsistent. Stating it differently in logical terms, the negation of a negative case must follow from **KB**. In general, test cases can be either complete or incomplete. A complete test case comprises all relevant parts (such as components and connections) of a configuration whereas an incomplete test case specifies only parts of a configuration, which in case of a positive test case must be expandable to a complete configuration. Conversely, a partial negative test case declares a partial configuration which cannot be expanded to a complete configuration. Configurations as defined in Section Chapter **??** are complete. A test case is a logical sentence.

Let us consider the following example of a positive incomplete test case c^+, which specifies that there must exist a configuration where a `CPUD` and a `CPUS` are on the same motherboard.

c^+ : $\texttt{MB}(m1) \wedge \texttt{CPUD}(c1) \wedge \texttt{CPUS}(c2) \wedge \texttt{cpu-of-mb}(c1,m1) \wedge \texttt{cpu-of-mb}(c2,m1)$.

quantified. The symbol `false` expresses unsatisfiability.

A negative test case c^- on the other hand specifies that a CPUD and a CPUS must not be mixed on silver and diamond motherboards.

c^- : $\texttt{CPUD}(c1) \wedge \texttt{CPUS}(c2) \wedge (\texttt{MBSilver}(m1) \vee \texttt{MBDiamond}(m1)) \wedge$
$\texttt{cpu-of-mb}(c1,m1) \wedge \texttt{cpu-of-mb}(c2,m1)$.

Testing the given knowledge base with these two test cases results in an undesired contradiction, i.e., $\mathbf{KB}_{faulty} \cup c^+$ is inconsistent. In particular, Axioms 7, 8, and 9 are unsatisfiable when combined with c^+. Consequently, one of these axioms must be changed when assuming that the test cases are correct. Note that apart from the faulty Axiom 7 also the correct Axioms 8 and 9 are involved in the contradiction. Therefore, at first sight one of these axioms might also be faulty. However, it will turn out that if we assume either Axiom 8 or 9 to be faulty, there must be an additional faulty axiom.

Let us first assume that Axiom 8 is faulty and all other axioms are correct. Axioms 7 and 9 combined with c^+ imply that motherboard $m1$ is of type silver. Consequently, Axioms 7, 9 and case c^+ imply c^-. Remember, however, that c^- must be inconsistent with the knowledge base. In other words, the knowledge base has to imply $\neg c^-$. Consequently, if we consider Axiom 8 as faulty and all other axioms of $\mathbf{KB}_{faulty}$ as correct, there is no way to extend this set of axioms to an acceptable configuration knowledge base given the test cases. Any change of $\mathbf{KB}_{faulty}$ where only Axiom 8 is deleted or replaced cannot be consistent with the positive case and inconsistent with the negative case. Similar arguments hold for Axiom 9.

It follows that only Axiom 7 is a single fault diagnosis of $\mathbf{KB}_{faulty}$ in the sense that changing Axiom 7 will allow the formulation of a knowledge base such that the positive case is consistent and the negative case is inconsistent. If either Axiom 8 or 9 is subject to a revision, additional axioms must be altered.

3.2 Defining Configuration and Diagnosis

The task of debugging a configuration knowledge base is to discover those axioms of **KB** which have to be changed. In particular, some of the axioms of a faulty knowledge base must be removed and possibly some axioms must be added. This extension is denoted by **EX**. We call such a revised knowledge base the target knowledge base. This target knowledge base has to be consistent with all positive test cases and must be inconsistent with all negative test cases. We start with the definition of the instances of a configuration knowledge base diagnosis problem (CKB-diagnosis problem).

Definition (Instances of a CKB-diagnosis problem) *A problem instance* $\langle \mathbf{KB}, \mathbf{C}^+, \mathbf{C}^- \rangle$ *of a configuration knowledge base diagnosis problem is defined by a set of logical sentences* **KB** *representing the configuration knowledge base, a set of positive test cases* $\mathbf{C}^+$ *and a set of negative test cases* $\mathbf{C}^-$*. The elements of* $\mathbf{C}^+, \mathbf{C}^-$ *are logical sentences representing test cases.*

Based on the instances of a CKB-diagnosis problem we define a diagnosis as a subset of the configuration knowledge base:

Definition (Diagnosis) *A set* $\mathbf{S} \subseteq \mathbf{KB}$ *is a diagnosis for a CKB-diagnosis problem instance* $\langle \mathbf{KB}, \mathbf{C}^+, \mathbf{C}^- \rangle$ *iff there exists a set of logical sentences* $\mathbf{EX}$ *extending* $\mathbf{KB}$ *such that*

- $(\mathbf{KB} - \mathbf{S}) \cup \mathbf{EX} \cup \{c^+\}$ *consistent* $\forall c^+ \in \mathbf{C}^+$ *and*
- $(\mathbf{KB} - \mathbf{S}) \cup \mathbf{EX} \cup \{c^-\}$ *inconsistent* $\forall c^- \in \mathbf{C}^-$

A diagnosis $\mathbf{S}$ defines those sentences of $\mathbf{KB}$ which must be deleted such that all positive examples are configurations or can be extended to configurations [6]. However, by deleting axioms the logical theory becomes weaker such that possibly some negative test cases are not inconsistent with $(\mathbf{KB} - \mathbf{S})$. Consequently, on one hand axioms are deleted from $\mathbf{KB}$ such that all positive test cases are consistent but on the other hand axioms must be added such that all negative test cases are inconsistent with the resulting knowledge base $(\mathbf{KB} - \mathbf{S}) \cup \mathbf{EX}$. If for a diagnosis $\mathbf{S}$ such an extension $\mathbf{EX}$ does not exist, the diagnosis $\mathbf{S}$ must be rejected. However, how can we verify that such an extension does not exist?

Since the negation of the negative test cases has to follow from $(\mathbf{KB} - \mathbf{S}) \cup \mathbf{EX}$, we can replace $\mathbf{EX}$ by the conjunction of the *negated* negative test cases. Note that we can exploit the usual formulation of constraints in logic-programming by employing the special symbol `false` to deal with negation. For example we can negate and generalize the negative test case of our example by:

$\neg c'^-$: $\texttt{CPUD}(X) \wedge \texttt{CPUS}(Y) \wedge (\texttt{MBSilver}(Z) \vee \texttt{MBDiamond}(Z)) \wedge$
$\texttt{cpu-of-mb}(X,Z) \wedge \texttt{cpu-of-mb}(Y,Z) \rightarrow \texttt{false}$.

If the negative test case is not considered in our example, Axiom 8 would have been a diagnosis. However, if the negative test case is taken into account, Axiom 8 cannot be a diagnosis since there is no way to extend $\mathbf{KB}_{faulty} - \{\text{Axiom } 8\}$ such that c^+ is consistent and c^- is inconsistent with the resulting knowledge base. The same argument holds for Axiom 9.

Proposition *The set* $\mathbf{S} \subseteq \mathbf{KB}$ *for a CKB-diagnosis problem instance* $\langle \mathbf{KB}, \mathbf{C}^+, \mathbf{C}^- \rangle$ *is a diagnosis iff* $\forall c^+ \in \mathbf{C}^+ : (\mathbf{KB} - \mathbf{S}) \cup \{c^+ \wedge \bigwedge_{c^- \in \mathbf{C}^-} (\neg c^-)\}$ *is consistent.*

It follows that a diagnosis will always exist under the (reasonable) assumption that positive and negative test cases do not interfere with each other.

Proposition *A diagnosis* $\mathbf{S}$ *for a CKB-diagnosis problem instance* $\langle \mathbf{KB}, \mathbf{C}^+, \mathbf{C}^- \rangle$ *exists iff* $\forall c^+ \in \mathbf{C}^+ : c^+ \wedge \bigwedge_{c^- \in \mathbf{C}^-} (\neg c^-)$ *is consistent.*

[6] Diagnosis S is denoted as Δ in Chapter **??**.

Usually, when debugging knowledge bases, the knowledge engineer is interested in minimal changes. This can be achieved by providing minimal diagnoses. A *minimal diagnosis* for the CKB-diagnosis problem instance $\langle \mathbf{KB}, \mathbf{C^+}, \mathbf{C^-} \rangle$ is a diagnosis $\mathbf{S}$ where no proper subset of $\mathbf{S'}$ of $\mathbf{S}$ is a diagnosis. In general, it is possible for multiple minimal diagnoses to exist, though additional criteria can be brought to bear to rank them in computation, as discussed below.

3.3 Computing Diagnoses

The above definitions allow us to employ the standard algorithms for consistency-based diagnosis with appropriate extensions for the domain. In particular, Reiter's Hitting Set algorithm (Reiter, 1987) can be applied which is based on the concept of conflict sets. Conflict sets provide an effective mechanism for focusing the search for diagnoses. The basic idea is to compute minimal sets of inconsistent logical sentences which are termed conflict sets. In order to resolve the inconsistency at least one sentence in each conflict set has to be changed. We assume $\mathbf{C^+}$ to be nonempty.

Definition (Conflict set) *A conflict set* $\mathbf{CS}$ *for* $\langle \mathbf{KB}, \mathbf{C^+}, \mathbf{C^-} \rangle$ *is a subset of* $\mathbf{KB}$ *such that* $\exists c^+ \in \mathbf{C^+} : \mathbf{CS} \cup \{c^+ \wedge \bigwedge_{c^- \in \mathbf{C^-}} (\neg c^-)\}$ *is inconsistent.*

A conflict set is minimal if it does not contain a proper subset which is a conflict set. The relation between diagnoses and conflict sets can be expressed by the concept of hitting sets. Given a set of sets $\overline{\mathbf{C}}$, a set $\mathbf{S}$ is a hitting set of $\overline{\mathbf{C}}$ iff $\mathbf{S} \cap \mathbf{C_i} \neq \emptyset$ for all $\mathbf{C_i} \in \overline{\mathbf{C}}$ and $\mathbf{S} \subseteq \bigcup_{\mathbf{C_i} \in \overline{\mathbf{C}}} \mathbf{C_i}$. A hitting set is minimal if it does not contain a proper subset which is a hitting set. In our domain the set $\overline{\mathbf{C}}$ corresponds to the set of conflict sets and a hitting set $\mathbf{S}$ represents a diagnosis.

Proposition *The set of logical sentences* $\mathbf{S}$ *is a minimal diagnosis for the CKB-diagnosis problem instance* $\langle \mathbf{KB}, \mathbf{C^+}, \mathbf{C^-} \rangle$ *iff* $\mathbf{S}$ *is a minimal hitting set for the set of all minimal conflict sets of* $\langle \mathbf{KB}, \mathbf{C^+}, \mathbf{C^-} \rangle$.

In (Felfernig et al., 2004) a method for diagnosing configuration knowledge bases is proposed which is based on a combination of Reiter's HS-DAG algorithm (Reiter, 1987; Greiner et al., 1989) and a divide-and-conquer approach to compute minimal conflict sets (Junker, 2004). The method is generally applicable to all knowledge-representation paradigms which are based on monotonic semantics such as predicate logic, description logic, and constraints. The approach requires a complete and correct consistency-checker which is employed to compute minimal conflict sets. Reiter's algorithm generates minimal diagnoses in a breadth-first manner, i.e., diagnoses with smaller cardinalities are constructed first. This is helpful to control computation costs since, in practice, it is sufficient for the knowledge engineer to compute some of the most probable diagnoses. These diagnoses are then used to generate additional tests and it is therefore not necessary to compute all minimal diagnoses (Shchekotykhin et al., 2012). In case it is more probable that logical sentences in a knowledge base are correct than faulty, the set of minimal cardinality diagnoses (i.e., those diagnoses where the number of elements is minimal) serves as an approximation for a set of the most probable diagnoses. Furthermore, if we limit the search for diagnoses to minimal cardinality diagnoses then the depth of the search space corresponds exactly to the number of elements in a minimal cardinality diagnosis. Since in practice the number of faulty axioms in a knowledge base is low, the application of a breadth-first strategy is feasible. In Chapter **??** techniques and algorithms for computing conflicts and diagnoses are presented.

Let us now reconsider our example knowledge base. In case only c^+ is provided there is one minimal conflict set $\{7, 8, 9\}$ comprising a minimal set of axioms which is inconsistent with c^+. Consequently, the minimal diagnoses are the axioms 7, 8, and 9 which are the minimal hitting sets of the minimal conflict set. Note that there is only one minimal conflict set in this case. If we consider the negative test case c^- (i.e., add $\neg c'^-$ to the knowledge base) then there is again only one minimal conflict set $\{7\}$. Consequently, there is exactly one minimal single-fault diagnosis $\{7\}$.

Alternatively, let us consider a different case and assume that Axiom 7 is wrongly defined as follows:

(7) $\texttt{MB(X)} \rightarrow \texttt{MBSilver}(X) \wedge \texttt{MBDiamond}(X) \wedge \texttt{MBUniversal}(X)$.

In this case the universal motherboard was not forgotten but the knowledge engineer mixed up $\vee$ and $\wedge$. Furthermore, let us assume that only the positive example c^+ is given. In this case the set of minimal conflict sets is $\{7, 4\}$, $\{7, 5\}$, $\{7, 6\}$, $\{7, 8\}$, and $\{7, 9\}$. Consequently, the minimal diagnoses (or hitting sets of the conflict sets) are $\{7\}$ and $\{4, 5, 6, 8, 9\}$.

To summarize, we presented a generic debugging approach for a declarative representation of configuration knowledge that employs model-based diagnosis techniques. In analogy to regression testing procedures in software engineering positive and negative examples for configured product instances are used to locate faults in a configurator application. The presented approach supports knowledge engineers by pointing them to minimal subsets of the knowledge base that are most probably faulty in case the configurator does not accept all the positive examples or does not reject all the negative examples.

Overall, the test cases (examples) thus play a central role in the approach and the availability of a sufficient amount of test cases can help to strongly reduce the number of diagnosis candidates. A typical point in time when errors are introduced into knowledge bases, is when the rules are updated or extended, which happens quite frequently in many application domains. In that case, all past configurations that have been calculated before the maintenance operation and which should be still valid, can thus serve as positive test cases and can therefore be used in regression tests. Furthermore, remember that the presented approach also supports the use of partial test cases, which means that the knowledge engineer can specify valid configuration fragments and is not required to manually check a complete and possibly complex configuration for correctness before defining it as a test case. Still, more support in the area of test case management is possible, e.g., through the automatic generation of representative test cases. We see this aspect as a relevant research direction not only in the configuration domain, but also for knowledge-based systems in general.

4 Organizational Challenges

The challenge of developing a configurator application has the setup and maintenance of a configuration knowledge base at its heart. However, beside the technical issues of knowledge encoding, acquisition of configuration knowledge from domain experts as well as its validation and maintenance that we discussed up to now, i.e., the *content* side, a variety of further *organizational*, *business*, and *process* facets is related to configuration knowledge base development. The knowledge base defines the variability of the configurable product offer in question. Initially, a company that is in the situation of configuration knowledge base development can either progress from, for instance, Mass Production (see Chapter **??**) or rework an already existing product configuration approach. While in the latter case organizational experience with Mass Customization (see Chapter **??**) and configurable product offerings is already in place, in the first case, knowledge base development will be accompanied by a variety of organizational and managerial issues. Usually, the product offering itself and the degree of variability that is offered for customization needs to be designed from scratch as the implementation of a Mass Customization strategy requires a fundamental rethinking of the product architecture itself. (Mikkola, 2009), for instance, explores the relationship between product architecture design and Mass Customization. She discusses a modularity measure for product architectures and the associated opportunities for Mass Customization strategies. Based on Mikkola's recommendations, different variants of configuration models can be (partially) assessed with respect to their aptness for successful Mass Customization strategies.

The process of product development itself typically consists of the specification of multiple views, in particular the functional, the technical, and the physical view (Jiao and Tseng, 1999; Arana et al., 2007). Another example of structuring configuration models along different views comes from (Malmgren et al., 2010) for the construction industry. Due to the two-phase construction of pre-fabricated buildings (production and assembly at the customer site) the physical view in (Jiao and Tseng, 1999) is divided into separate production and assembly views. This modularization of configuration models is particularly necessary in case of a graphical representation of the configuration model in order to reduce diagrammatic complexity and to avoid conceptual overload of the involved stakeholders (Moody, 2009). The functional view models the functionality and typically depends on the analysis of customer requirements in target market niches. In contrast, the technical view *describes the product design by its modules and modular structure* and ensures technological feasibility by enforcing design parameters and structuring mechanisms. In the third step – the physical view – the product design has to be mapped to physical components and assemblies. There, the manufacturability of the configured products as well as costs, volumes, and schedules are in the focus and can lead to feedback loops to functional and technical views. Obviously, these process steps can therefore only be executed in an iterative and interleaved manner where different organizational units like marketing & sales, design, and manufacturing & logistics collaborate tightly. In (Jiao and Tseng, 1999) a methodology for developing a product family architecture for Mass Customization based on these three process steps is proposed and empirical results from a case study are provided for further reading. In addition, (Nellore et al., 1999) focus on the initial specification process in new product development and propose a model to manage the interaction of the different stakeholders in this early and vital phase.

While the presented UML-based approach is very helpful in easing communication between the different stakeholders in a configurator development project, further means of external documentation of configuration knowledge have been explored. (Haug, 2010) generalizes the idea of configurator documentation systems and proposes a product model management software that constitutes a knowledge management and learning platform for the stakeholders involved in a product configuration development and maintenance project. The documentation software supports representation concepts such *class information cards* and *product variant master* (PVM) that are adequately comprehensible to domain experts. Haug informedly argues that cutting cost on documentation in configurator development projects is short-sighted and leads to reduced or no maintainability of the developed configurator applications. Finally, the book of (Hvam et al., 2008) discusses several case studies on the implementation of Mass Customization strategies with the help of product configuration systems from a practical point of view and can therefore also serve as a helpful guide.

5 Conclusions and Further Reading

Most modern and non-trivial product configuration systems rely on a knowledge base that restricts the solution space of potentially configured products to those instances that are actually valid according to technical or business-related restrictions and constraints. The goal of this chapter was to elaborate on the process from acquiring these "technical and economic restrictions and constraints" from their sources as well as encode, maintain, and validate it during the operational lifetime of the configuration system. Initially, we gave a rough overview on the development from the first rule-based knowledge representation mechanisms to a logic-based encoding that follows the component-port paradigm. In addition, we presented a UML-based representation mechanism for configuration knowledge acquisition. Such visual models abstract from the concrete declarative representation mechanism and allow domain experts a more intuitive access to knowledge base development in analogy to model-driven approaches in software engineering. However, in this chapter we only focused on planar representations of configuration models and did not address the concept of *views* as a structuring mechanism. An extension to the presented UML-based approach (Felfernig et al., 2000b) that included a functional view for specifying abstract user requirements as opposed to a unified technical and physical view was proposed in (Felfernig et al., 2000c). Other extensions include representation of personalization knowledge in order to ease self-service configuration scenarios in e-commerce (Ardissono et al., 2003; Tiihonen and Felfernig, 2010) or to facilitate personalized optimization functions for computing configured instances (Zanker et al., 2010). Another aspect of knowledge engineering for configurator development addresses the integration with partners in supply networks. In cases where there are organizational requirements to keep detailed configuration knowledge hidden from other participants in a supply chain, a distributed solving mechanism needs to be applied (Felfernig et al., 2001). In (Jannach and Zanker, 2013) alternate modeling approaches for such a distributed configuration problem based on a CSP-based formalism are discussed and different distributed solving mechanisms comparatively assessed. A more abstract discussion on the perspectives of Mass Customization in dynamic value networks can be found in (Dietrich and Kim, 2006). Based on a case study of custom-made shoe production, opportunities for adaptable business models are explored.

The validation of configuration knowledge bases by employing model-based diagnosis techniques is another key aspect of this chapter. With the presented approach minimal sets of potentially faulty statements in a declarative knowledge can be identified when sets of positive and negative examples of configured product instances are given. As has been pointed out, the maintenance and evolution of configuration knowledge bases is greatly facilitated by enabling such a form of regression testing. We refer to Felfernig et al. (Felfernig et al., 2004) for technical details of this debugging approach. Another opportunity for the employment of model-based diagnosis techniques is the reconfiguration problem. As opposed to computing a configured product instance from scratch, in a reconfiguration task an already configured instance has to be modified in order to fulfill new or changed customer requirements or to be consistent with a modified version of a configuration knowledge base. Efficient support for such an *after-sales* service is still an open challenge. See (Jannach et al., 2007) for a comprehensive survey on these research issues in knowledge-based configuration.

Finally, some of the organizational challenges in the configuration domain have also been addressed in this chapter. The implementation of a Mass Customization strategy goes well beyond the engineering effort of a configurator application itself. For instance, (Heiskala et al., 2007) provide a comprehensive literature survey on the implementation processes of Mass Customization strategies and their associated configurator development. The construction of configuration knowledge bases is closely related to new product development and product design in general (Otto and Wood, 2001) and to the creation of a product architecture for Mass Customization in particular (Mikkola, 2009). Therefore, the knowledge engineering effort for configurator development needs to be closely integrated into these processes, and there must be a clear understanding of the interactions between the different stakeholders that also serve as sources for configuration knowledge (Jiao and Tseng, 1999; Nellore et al., 1999).

References

Agarwal, R., Tanniru, M., 1992. A structured methodology for developing production systems. Decision Support Systems 8 (6), 483 – 499.

Arana, J., Elejoste, M., Lakunza, J. A., Uribetxebarria, J., Zangitu, M., 2007. Mass Customization Information Systems in Business. Idea, Ch. Product Modeling and Configuration Experiences, pp. 33–58.

Ardissono, L., Felfernig, A., Friedrich, G., Goy, A., Jannach, D., Petrone, G., Schäfer, R., Zanker, M., 2003. A framework for the development of personalized, distributed web-based configuration systems. AI Magazine 24 (3), 93–108.

Barker, V., O'Connor, D., Bachant, J., Soloway, E., 1989. Expert systems for configuration at digital: Xcon and beyond. Communications of the ACM 32 (3), 298–318.

Dietrich, A. J., Kim, S., 2006. Implications of mass customization on business information systems. International Journal on Mass Customization 1 (2/3), 218–236.

Felfernig, A., Friedrich, G., Jannach, D., 2000a. Generating product configuration knowledge bases from precise domain extended uml models. In: Proceedings 12th International Conference on Software Engineering and Knowledge Engineering. Chicago, USA, pp. 284–293.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., February 2004. Consistency-based diagnosis of configuration knowledge bases. Artificial Intelligence 152, 213–234.

Felfernig, A., Friedrich, G., Jannach, D., Zanker, M., 2001. Towards distributed configuration. In: Proceedings of the 24th German Conference on Artificial Intelligence (KI). Vol. 2174. Springer, pp. 198–212.

Felfernig, A., Friedrich, G., Jannach, D., Zanker, M., 2002. Web-based configuration of virtual private networks with multiple suppliers. In: Proceedings 7th International Conference on Artificial Intelligence in Design (AID-02). Kluwer Academic Publisher, Cambridge, UK, pp. 41–62.

Felfernig, A., Jannach, D., Zanker, M., 2000b. Contextual diagrams as structuring mechanisms for designing configuration knowledge bases in uml. In: 3rd International Conference on the Unified Modeling Language (UML2000). Vol. 1939 of Springer Lecture Notes in Computer Science. pp. 240–254.

Felfernig, A., Jannach, D., Zanker, M., 2000c. Contextual diagrams as structuring mechanisms for designing configuration knowledge bases in UML. In: Proceedings of the 3rd International Conference on the Unified Modeling Language (UML). Springer, pp. 240–254.

Fleischanderl, G., Friedrich, G. E., Haselb"ock, A., Schreiner, H., Stumptner, M., Jul./Aug. 1998. Configuring large systems using generative constraint satisfaction. IEEE Intelligent Systems 13 (4), 59–68.

Fowler, M., 2010. Domain-Specific Languages. Addison-Wesley.

Greiner, R., Smith, B. A., Wilkerson, R. W., 1989. A Correction to the Algorithm in Reiter's Theory of Diagnosis. Artificial Intelligence 41 (1), 79–88.

Haug, A., 2010. A software system to support the development and maintenance of complex product configurators. International Journal of Advanced Manufacturing Technology 49, 393–406.

Heiskala, M., Tihonen, J., Paloheimo, K.-S., Soininen, T., 2007. Mass Customization Information Systems in Business. Idea, Ch. Mass Customization with Configurable Products and Configurators, pp. 1–32.

Hvam, L., Mortensen, N. H., Riis, J., 2008. Product Customization. Springer.

Jannach, D., Felfernig, A., Kreutler, G., Zanker, M., Friedrich, G., 2007. Mass Customization Information Systems in Business. Idea, Ch. Research Issues in Knowledge-Based Configuration, pp. 221–236.

Jannach, D., Zanker, M., 2013. Modeling and solving distributed configuration problems: A csp-based approach. IEEE Transactions on Knowledge and Data Engineering 25 (3), 603–618.

Jiao, J., Tseng, M. M., 1999. A methodology of developing product family architecture for mass customization. Journal of Intelligent Manufacturing 10, 3–20.

Junker, U., 2001. Preference-based programming for Configuration. In: Proceedings of the Workshop on Configuration held in conjunction with IJCAI'01. Seattle, WA.

Junker, U., 2004. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In: 19th Intl. Conference on Artifical Intelligence. AAAI'04. AAAI Press, pp. 167–172.

Junker, U., Mailharro, D., 2003. Preference programming: Advanced problem solving for configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AIEDAM) 17 (1), 13–29.

Mailharro, D., 1998. A Classification and Constraint-based Framework for Configuration. AI EDAM 12 (04), 383–397.

Malmgren, L., Jensen, P., Olofsson, T., 2010. Product modeling of configurable building systems - a case study. Journal of Information Technology in Construction 16, 697–712.

McDermott, J., 1982. R1: A Rule-based Configurer of Computer Systems. Artificial Intelligence Journal 19, 39–88.

McGuinness, D., Wright, J., Jul/Aug 1998. An Industrial-Strength Description Logic-Based Configurator Platform. IEEE Intelligent Systems 98, 69–77.

Mikkola, J. H., 2009. Management of product architecture modularity for mass customization: Modeling and theoretical considerations. IEEE Transactions on Engineering Management 54 (1), 57–69.

Mittal, S., Frayman, F., 1989. Towards a Generic Model of Configuration Tasks. In: Proc. of Eleventh Int. Joint Conf. on AI IJCAI-89. Detroit, Michigan, USA, pp. 1395–1401.

Moody, D. L., 2009. The "Physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Software Eng. 35 (6), 756–779.

Nellore, R., Söderquist, K., Eriksson, K.-A., 1999. A specification model for product development. European Management Journal 17 (1), 50–63.

Otto, K. N., Wood, K. L., 2001. Product Design. Prentice Hall.

Reiter, R., 1987. A theory of diagnosis from first principles. AI Journal 23 (1), 57–95.

Shchekotykhin, K. M., Friedrich, G., Fleiss, P., Rodler, P., 2012. Interactive ontology debugging: two query strategies for efficient fault localization. J. Web Sem. to appear.

Steve, J., Davis, 1990. Effect of modularity on maintainability of rule-based systems.

International Journal of Man-Machine Studies 32 (4), 439 – 447.

Studer, R., Benjamins, V. R., Fensel, D., 1998. Knowledge engineering: Principles and methods. Data and Knowledge Engineering 25 (1-2), 161–197.

Tiihonen, J., Felfernig, A., 2010. Towards recommending configurable offerings. International Journal of Mass Customization 3 (4), 389–406.

Zanker, M., Aschinger, M., Jessenitschnig, M., 2010. Constraint-based personalized configuring of product and service bundles. International Journal on Mass Customization 3 (4), 407–425.