

Florian Speiser-Reinfrank

Intelligent Supporting Technologies for the Maintenance of Product Configuration Systems

Doctoral Thesis

Graz University of Technology

Institute for Software Technology

Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Supervisors:

Univ.-Prof. Dipl.-Ing. Dr.techn. Alexander Felfernig and

Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Graz, October 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____
Date

Signature

Eidesstattliche Erklärung¹

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, _____
Datum

Unterschrift

¹ Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Product configuration and knowledge-based systems are commercially successful applications, because they are an excellent medium for bringing products and services of companies and customer's requirements together. Nowadays, they are used in many domains. The website <http://www.configurator-database.com/> gives an overview of commercially used product configuration systems and their underlying knowledge bases.

Such knowledge bases have to be maintained over time because new releases lead to new or updated features or customer preferences are changing over time. As stakeholders are not able to describe or communicate all changes correctly, new knowledge engineers do not have a correct understanding of the knowledge base, or many knowledge engineers are working on the same knowledge base at the same time, maintenance tasks can be time-consuming and error prone. Such difficulties can lead to a knowledge base which does not present the set of available variants and this may lead to a low customer satisfaction and frustration with the system.

This thesis picks up the research question how intelligent techniques can support knowledge engineers when they maintain a product configuration knowledge base. We show that such techniques can help to receive functional knowledge bases in terms of consistency, increase the understandability via reducing the content displayed to the knowledge engineer (e.g., using recommendation techniques), and optimize the maintainability of a knowledge base with an efficient anomaly management.

Kurzfassung

Produktkonfiguratoren und wissensbasierte Empfehlungssysteme sind kommerziell äußerst erfolgreich. Sie ermöglichen eine optimale Abstimmung von Produkteigenschaften und Dienstleistungen mit Kundenpräferenzen. Heutzutage wird diese Eigenschaft in vielen Domänen erfolgreich angewendet. Die Webseite <http://www.configurator-database.com/> gibt eine Auswahl von Online-Produktkonfiguratoren wieder.

Wissensbasen, die zur Repräsentation der zu konfigurierenden Produkte und deren Eigenschaften verwendet werden, müssen häufig gewartet werden, weil sich z.B. die Kundenanforderungen ändern oder die angebotenen Produkte neue Funktionen enthalten. Dabei können einige Probleme auftreten. Z.B. können Informationen von Stakeholdern falsch aufbereitet oder kommuniziert werden, neue Entwickler kennen die Wissensbasis noch nicht vollständig oder es arbeiten gleichzeitig mehrere Ingenieure an der gleichen Wissensbasis zur selben Zeit. Wenn ein Problem eintritt kann nicht sicher gestellt werden, dass die überarbeitete Wissensbasis die Realität widerspiegelt. Die Endanwender derartiger Systeme können wegen solcher Fehler schnell frustriert sein.

Diese Dissertation zeigt, wie intelligente Technologien verwendet werden können, um Ingenieure bei der Wartung von Konfigurationswissensbasen zu unterstützen. Wir verwenden u.a. Empfehlungssysteme, damit die verfügbaren Informationen in der Wissensbasis auf das Wesentliche reduziert und die Verständlichkeit erhöht wird. Außerdem verwenden wir Technologien, die Anomalien in der Wissensbasis nicht nur identifizieren, sondern auch erklären und reparieren können.

Acknowledgements

I want to thank the team of the Institute for Software Technology at the Graz University of Technology. Especially the discussions with Gerald Ninaus, Martin Stettinger, Stefan Reiterer, Michael Jeran, and Harald Grabner helped to get new ideas and evaluate existing research challenges. Franz Wotawa helped to finalize this thesis with interesting hints for several sections of this thesis. The administrative tasks related to this thesis would be irresolvable without the help of Petra Pichler. I also want to thank Alexander Felfernig and the co-authors of the papers which are the basis for this thesis.

Furthermore I also want to thank to my family for their great support. I am very thankful to my mother Cécilia and my wife Daniela for their encouragement and understanding. Last but not least I want to thank Waltraud Langmayr for her proof-reading of this thesis and Alexander Hardt-Stremayr for interesting discussions related to this thesis.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	2
1.3	Outline	5
2	Related Work	7
2.1	Basic Techniques for Product Configuration Systems	7
2.2	Types of Product Configuration Systems	13
2.3	Maintaining Product Configuration Systems	22
3	Constraint-based Product Configuration Systems	33
3.1	Constraint Satisfaction Problems	33
3.2	Definitions	38
3.3	Running Examples	44
4	Anomaly Management for Constraint-based Product Configuration Systems	55
4.1	Types of Anomalies in Constraint-based Product Configuration Systems	55
4.1.1	Conflicts and Diagnoses	55
4.1.2	Redundancies	57
4.1.3	Well-formedness Violations	59
4.2	Anomaly Explanation	60
4.3	Automated Repair of Scoring Rules	63
4.4	Visualization of Conflicts	76

Contents

5	Intelligent Supporting Techniques for Maintaining Constraint-based Systems	89
5.1	Recommendation Techniques	90
5.2	Anomaly Management	92
5.2.1	Inconsistencies	93
5.2.2	Redundancies	99
5.2.3	Well-formedness Violations	106
5.3	Dependency Detection	109
5.4	A Goal-Question-Metrics Model for Product Configuration Knowledge Bases	115
5.4.1	Goals for Product Configuration Knowledge Bases	115
5.4.2	Questions for Product Configuration Knowledge Bases	117
5.4.3	Metrics for Product Configuration Knowledge Bases	119
5.4.4	Discussion	124
5.5	Test Case Generation	128
5.6	Constraint-based Product Configuration system Development	131
6	Interfaces for the Maintenance of Constraint-based Systems	135
6.1	Current user Interfaces for the Maintenance of Constraint-based Systems	135
6.2	iCone: an Interface for the Maintenance of Constraint-based Systems	156
7	Conclusion	167
7.1	Summary	167
7.2	Further Research	169
	Bibliography	171

List of Figures

1.1	Intelligent support for the maintenance of product configuration knowledge bases	3
3.1	Different types of anomalies.	40
4.1	Graphical representation of all conflicts in Example 3.3 calculated with HSDAG and QuickXPlain.	57
4.2	Example financial service recommender applications	74
4.3	Notebook recommendation	78
4.4	Presentation of 1 to n diagnoses denoted as 'change recommendation'	79
4.5	Presentation of 1 to n conflicts	79
4.6	Presentation of 1 to n diagnoses and conflicts	80
4.7	Presentation of fitness values	80
4.8	Ranking of selected diagnosis / conflict	85
5.1	Number of consistency checks for calculating diagnoses	100
5.2	Number of consistency checks of redundancy detection algorithms	103
5.3	Graphical representation for detecting redundant assignments	107
5.4	Detection of redundant assignments	108
5.5	Visualization of changes for the metric <i>DEAD</i>	127
6.1	RECOMOBILE user interface for the representation of (adaptive) defaults.	146
6.2	Configuration comparison interface based on price ranking.	147
6.3	Configuration comparison interface based on utility-based ranking.	148
6.4	Configuration comparison interface based on utility-based ranking.	148

List of Figures

6.5	COMBEENATION: integrated development and visualization of configurators.	152
6.6	A prototype web-based bicycle configurator	156
6.7	Main screen of the icone interface	157
6.8	Presentation of constraints and recommendation techniques .	159
6.9	Presentation of anomalies in the iCone interface	161
6.10	Dependencies of constraints	162
6.11	Dependencies of domain elements between two variables . .	163
6.12	Visualization of metrics	165

List of Tables

2.1	Classification of configuration systems	14
2.2	Evaluation of constraint representations	17
2.3	Methods for KBS validation and timing in the life-cycle	29
2.4	Applicability of evaluation techniques to KBS development artifacts	31
3.1	Example customer variables	47
3.2	Scoring rules for customer variable <i>investment period</i>	48
3.3	Scoring rules for customer variable <i>goal</i>	48
3.4	Example customer requirements	49
3.5	Interests of customer <i>Robert</i>	49
3.6	Example set of financial services	50
3.7	Example product variables	50
3.8	Recommendation result	51
3.9	Scoring rules for product variable <i>shares</i>	51
3.10	Scoring rules for product variable <i>value fluctuation</i>	51
3.11	Product-specific scoring rules $\{up_1, \dots, up_6\}$	51
3.12	Product-specific scoring rules $\{up_7, \dots, up_{12}\}$	51
3.13	Example constraints	52
3.14	Utilities of products regarding interest dimensions	52
3.15	Utilities of products for customer <i>Robert</i>	52
3.16	Examples of intended service orderings.	52
4.1	Product configuration analysis operations, property checks, and related explanations	64
4.2	Utilities of products for customer <i>Robert</i>	73
4.3	Performance of the Minos solver	76
4.4	Overview of the user activities and scenarios	79
4.5	Inconsistency resolving time	83

List of Tables

4.6	Inconsistency repair time	84
4.7	Understandability of inconsistencies	86
4.8	Satisfaction with the product assortment	86
4.9	Satisfaction with the presented conflicts	87
5.1	Collaborative filtering example	91
5.2	Content-based recommendation example	92
5.3	Evaluation of FastDiag and HSDAG with the Car Selection FM	97
5.4	Evaluation of FastDiag and HSDAG with the SmartHome V 2.2 FM	98
5.5	Evaluation of FastDiag and HSDAG with the Xerox FM	99
5.6	Evaluation of CoreDiag with selected S.P.L.O.T. feature models	102
5.7	Relations between goals and metrics	118
5.8	Relations between metrics and questions	125
5.9	Anomaly calculation time	128
5.10	An example for randomly generated test cases.	129
5.11	An example case base for evaluating randomly generated test cases.	130
6.1	Example of determining relevant questions on the basis of <i>collaborative filtering</i>	143
6.2	Example of determining default values for the parameters of the computer configurator on the basis of <i>collaborative filtering</i>	145
6.3	Example list of user preferences.	145
6.4	Example list of product utilities.	147
6.5	Example of determining relevant constraints on the basis of <i>collaborative filtering</i>	150
6.6	Design principles of configurator user interfaces and techno- logical foundations.	154

List of Algorithms

1	QUICKXPLAIN(C_{KB}, C_R): Δ	93
2	QUICKXPLAIN'($C_{KB}, \Delta, C_R = \{c_1, \dots, c_r\}$): Δ	94
3	FASTDIAG(C_R, C): Δ	95
4	DIAG($\Delta, C_R = \{c_1, \dots, c_r\}, C$): Δ	96
5	SEQUENTIAL(C): Δ	101
6	COREDIAG (C): Δ	101
7	CORED($C_{KB}, \Delta, C_R = \{c_1, c_2, \dots, c_r\}$): Δ	102
8	AssignmentSequential	106
9	DeadDomainElement (V, D, C): Δ	109
10	FullMandatory (V, D, C): Δ	110
11	UnnecessaryRefinement (C, V): Δ	111
12	GibbsSampling	112

List of abbreviations

CBS	Constraint-based system
CSP	Constraint satisfaction problem
FM	Feature Model
FOL	First order logic
GQM	Goal, Questions, and Metrics
HSDAG	Hitting set directed acyclic graph
KB	Knowledge base
KBS	Knowledge-based system
MAUT	Multi attribute utility theory
OWL	Web Ontology Language
PCS	Product Configuration System
QX	QuickXPlain
UC	Utility constraint

*"Building Knowledge-based Systems
is something of an art."*

Preece, 1998

1 Introduction

1.1 Motivation

'I want to buy this car but the color should be black instead of blue.' 'The notebook should have 8GB RAM and not 4GB.' 'We have to change our production line s.t. we can also manufacture mountain bikes for children not only for adults.'

In today's world many products are highly configurable because the costs for offering such techniques have decreased, the usability of such systems has got better, and the number of people having internet access has also been increasing enormously for decades. Configurable products are characterized by specific product attributes which are comparable. Examples for configurable product domains are cars, notebooks, bikes, and financial services.

Mass customization supports customers getting their individual products based on a standardized basic model [Reichwald and Piller, 2009]. Especially for highly customizable products the purchase of such products increases when customers have greater confidence in their individual preferences [Fogliatto et al., 2012; Bharadwaj et al., 2009]. A customized product can be denoted as variant (economic term) or instance (technical term). Several techniques have been developed to support mass customization strategies. For example, constraint-based systems and configuration systems try to find a configuration for a product which satisfies as many customer preferences as possible [John and Geske, 2003]. The implementation process of a system, which detects optimal product configurations, is denoted as **knowledge**

1 Introduction

base development. The development process contains requirements specification, implementation, and testing of the new created knowledge base [Felfernig et al., 2014c; Studer et al., 1998].

While the process of creating a new knowledge base can be seen as a finite process with a starting (e.g., a kickoff meeting) and a finishing point (e.g., first productive usage of the new knowledge base), the requirements for such systems are changing over time. For example, new vendors for the customizable product lead to new specification of the products, new customer requirements lead to updates of the customer interface and the knowledge base, and new requirements from the marketing team (e.g., introducing mountain bikes for children) lead to new requirements. This means that we have to adapt knowledge bases with thousands of product attributes and the relations between these attributes. Such adaption processes are time consuming and error prone. Gartner [2005] defines ten risks when configuration knowledge bases will be introduced in companies. One of these risks is the **maintenance of the product configuration**. If the product knowledge base can not be updated correctly and in time, the system will not be fit to the customers' preferences.

1.2 Contributions

The maintenance task for product configuration systems subsumes the following steps. First, a concrete semi-formal specification is necessary. Therefore, the knowledge of all stakeholders for this maintenance task has to be considered. Second, influences on the existing knowledge base have to be observed and evaluated. Third, besides the functional requirements beyond the specification knowledge engineers also have to consider non-functional requirements like the performance of the system. If all changes of the updated knowledge base are considered, the knowledge engineer can bring the updates into the productive system. Since all of the tasks of knowledge engineers are very complex, we are answering the following research question in this thesis:

How can we support knowledge engineers in their product configuration knowledge base maintenance tasks?

To tackle this research issue we first analyzed the maintenance task and its issues. For example, the complexity and size of a knowledge base leads to unconsidered elements of a knowledge base and a misunderstanding of anomalies can lead to invalid updates of the knowledge base. To tackle this issues we take a look at possible intelligent support techniques for product configuration knowledge bases. An overview of these techniques is given in Figure 1.1.

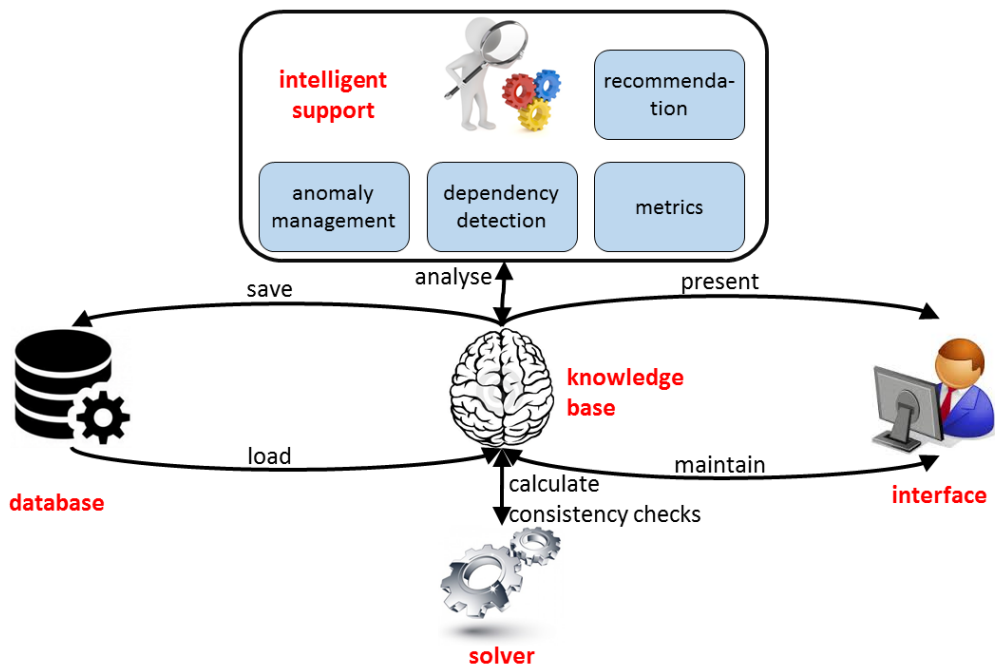


Figure 1.1: Intelligent support for the maintenance of product configuration knowledge bases. This thesis focuses on presenting recommendations, detecting anomalies and dependencies, and calculating metrics [Wotawa et al., 2015a].

1 Introduction

The figure consists of five main parts. First, the **knowledge base** itself is a representation interface of the configurable product in a formal representation (e.g., CSP). The knowledge base has to be **saved** (e.g., in a database) and a **solver** can determine, if a configuration is consistent. The results of the knowledge base - (in)consistent products and configurations - will be presented to customers and knowledge engineers (**interface**).

Finally, our research focuses on the *maintainability of constraint-based product configuration knowledge bases for knowledge engineers* and *adaptability of end-user preferences*. With this technique we can configure and rank products in constraint-based recommendations systems. Our research for **end-users** consist of the optimal presentation technique of the **no-solution-could-be-found-dilemma** in terms of satisfaction, time, and understandability.

For **knowledge engineers** we researched **anomaly management** which is the detection, presentation, explanation, and repair action of inconsistencies, redundancies, and well-formedness violations. *Recommendation systems* divide information into relevant and unnecessary information. We use this technique to present relevant information to knowledge engineers instead of the whole information within a knowledge base. To generate reports for variety management (e.g., 'How many bikes can be used for hill climbing?') we detect *dependencies* between product attributes. Finally, to predict the future *effort* for maintaining the knowledge base, we also introduced a GQM approach for product configuration knowledge bases to evaluate the quality of a knowledge base in terms of understandability, maintainability, and functionality.

With these techniques we help knowledge engineers in their maintenance tasks. This thesis shows how we can increase the understandability and maintainability of knowledge bases by using the techniques depicted in Figure 1.1.

1.3 Outline

Chapter 2 gives an overview of current **research** in the context of product configuration. Different concepts and types of product configuration systems are presented in Section 2.1. An overview of related work in the context of recommendation techniques [Felfernig et al., 2013b], basic approaches for the detection of inconsistencies [Felfernig et al., 2013c, 2014e], the state-of-the-art of simulation techniques, and related research in the context of knowledge base evaluation is given in Section 2.3.

The next Chapter 3 describes **constraint-based product configuration systems**. Therefore, we first describe principles of constraint satisfaction problems (Section 3.1; Reinfrank et al. [2015a]). Section 3.2 introduces basic definitions which will be used in this thesis [Reinfrank et al., 2015a]. To explain the techniques, which are presented in this thesis, we introduce running examples in Section 3.3.

The term **anomaly management** will be explained in detail in Chapter 4. The different types of anomalies - inconsistencies (Section 4.1.1; Felfernig et al. [2014e]), redundancies (Section 4.1.2; Felfernig et al. [2014d]), and well-formedness violations (Section 4.1.3; Felfernig et al. [2013a]; Benavides et al. [2013, 2010]) - are explicated in detail in Section 4.1. How an anomaly can be explained to a user is described in Section 4.2 [Felfernig et al., 2013a]. Possible repair actions for inconsistencies will be listed in Section 4.3 [Felfernig et al., 2013c]. Finally, Section 4.4 explains how inconsistencies can be presented to end-users and it shows the influences of the representation regarding understandability and satisfaction with the presentation [Wotawa et al., 2015b].

Intelligent supporting techniques for knowledge engineers are described in Chapter 5. Based on the description in Figure 1.1, we first explain how we can divide the information within a knowledge base into relevant and

1 Introduction

unnecessary information for a specific maintenance task (Section 5.1; Felfernig et al. [2013b]). Section 5.2 describes techniques to find anomalies in a product knowledge base [Reinfrank et al., 2015a; Felfernig et al., 2012a; Schubert et al., 2011]. We give a brief information about current inconsistency and diagnosis algorithms, explain current and new redundancy detection algorithms, and introduce algorithms for finding well-formedness violations. How simulation techniques can be used to detect dependencies between variables, is explained in Section 5.3 [Reinfrank et al., 2015a]. To measure the quality of a knowledge base, we describe a GQM approach for evaluating a product configuration knowledge base (Section 5.4; Reinfrank et al. [2015b]). A technique for automated test case generation and collaborative verification of these test cases is presented in Section 5.5 [Reinfrank et al., 2015c]. Besides the support for knowledge base maintenance tasks we give a short overview of principles for product configuration knowledge base development in Section 5.6 [Reinfrank et al., 2015c].

Apart from the calculation of anomalies and the implementation of intelligent supporting techniques for knowledge engineers the presentation of the information also plays an important role. Chapter 6 describes **interfaces for knowledge-based product configuration systems**. Section 6.1 gives an overview of current interfaces for such systems and describes requirements regarding such systems [Felfernig et al., 2014a]. In Section 6.2 we introduce our new web-based maintenance tool for product configuration knowledge base and show how we have implemented the intelligent supporting techniques for knowledge engineers presented in this thesis in our system [Wotawa et al., 2015a].

Finally, Chapter 7 concludes this thesis with a summary (Section 7.1) and gives hints for future research (Section 7.2) in the context of product configuration knowledge bases.

2 Related Work

There is a long history of research on modeling constraints and improving knowledge engineering processes. Early research focused on model-based knowledge representations that allowed a separation of domain and problem solving knowledge. An example of such a representation is the constraint presentation which became extremely popular as a technological basis for industrial applications [Freuder, 1997]. As the next step, graphical knowledge representations [Felfernig et al., 2000] and intelligent techniques for knowledge base testing and debugging have been developed [Felfernig et al., 2004a]. The following sections describe the state-of-the-art in research focusing on representing knowledge with constraint-based technologies and supporting techniques for knowledge base maintenance.

2.1 Basic Techniques for Product Configuration Systems

First, this section gives an overview about different concepts of product configuration systems in general. Thereafter three basic techniques for commercially successful configuration systems will be described: constraint-based product configuration (one product can be customized), knowledge-based product recommendation (many products can be compared based on their quantitative product attributes, e.g., price), and MAUT (multi attribute utility theory; many products can be compared based on their qualitative product attributes, e.g., risk).

Different Conceptualizations of Product Configuration Systems¹

The artificial intelligence community addresses a software tool when speaking about configuration systems. Bourke [2000] defines a product configuration system as '*...software modules with logic capabilities to create, maintain, and use electronic product models that allow complete definition of all possible product option and variation combinations, with a minimum of data entries and maintenance*'. The main technical component of the configuration system is the knowledge base which consists of the database and configuration logic. Whereas the database contains the total set of component types and their instances, the configuration logic specifies the set of restrictions on how components can be combined. In the following, different classifications of product configuration systems are presented.

Classification according to the configuration knowledge. The conceptualizations of configuration knowledge can be classified to (a) rule-based, (b) case-based and (c) model-based approaches. Each approach relies on a different ontology that is required to represent the domain knowledge and describe the object types (variables and domain elements) and the relations among object instances (constraints) [Sabin and Weigel, 1998].

- Rule-based approach: rule-based configuration systems work by executing rules with the following form: "if condition then consequence". The product solutions are derived in a forward-chaining manner. At each step, the system examines the entire set of rules and considers only the set of rules that can be executed next. Furthermore, there is no separation between directed relationships and actions. Thus, rules contain both the domain knowledge such as compatibilities between components and the control strategy that is necessary to compute the solution to a specific configuration problem [Sabin and Weigel, 1998]. The main drawbacks of rule-based systems are ascribed to the problems encountered during knowledge acquisition, consistency checking, knowledge maintenance, etc. [Günter and Kühn, 1999].

¹This section is taken from Blecker et al. [2004].

2.1 Basic Techniques for Product Configuration Systems

- Case-based approach: case-based reasoning relies on the assumption that similar problems have similar solutions. The required knowledge for reasoning consists of cases that record a set of product configurations accepted by earlier customers. The current configuration problem is solved by finding and adapting a previous solution to similar customer requirements.
- Model-based approach²: The most important model-based representation for product configuration systems are: logic-based, resource-based and **constraint-based** approaches [Sabin and Weigel, 1998]. The most prominent family of logic-based approaches is based on description logic. Description logics are formalisms for representing and reasoning with knowledge. They are based on the notions of individuals (objects), concepts (unary predicates, classes), roles (binary relations) and constructors that allow complex concepts and roles to be built from atomic ones. The inference mechanism is based on subsumption. However, resource-based systems are based upon a producer-consumer model of the configuration task. Each technical entity is characterized by the amount of resources it supplies, uses and consumes. A product configuration is acceptable when a resource balancing is realized [Juengst and Heinrich, 1998]. In constraint-based reasoning components are defined by a set of variables and their domain elements. Constraints among domain elements restrict the ways components can be combined [Tsang, 1993]. For example, a restriction can forbid a combination of parts (domain element $dom_1 \in dom(v_a)$ cannot be combined with $dom_2 \in dom(v_b)$) or can require a specific combination ($dom_1 \in dom(v_a)$ requires $dom_2 \in dom(v_b)$).

Classification according to the business strategy. From the point of view of mass customization, three main strategies with different requirements on product configuration systems are distinguished, namely **assemble-to-order**, **fabricate-to-order** and **engineer-to-order**. The assemble-to-order concept enables customers to configure a product by combining a finite number of standard domain elements. However, fabricate-to-order and engineer-to-order may assume an infinite number of configuration possibilities. The

²The next section describes this technology in more detail.

2 Related Work

technical realization of configuration systems for fabricate- and engineer-to order is more demanding than those for assemble-to-order because a parametrization of component dimensions should be made possible.

Classification according to organization. The organization of a configuration system can be either central or distributed. A central configuration system works locally and its configuration knowledge is completely stored in one unique system. All potential product instances that may represent a solution to the customer configuration problem are derived from this local data and one knowledge engineer maintains this knowledge base. However, the knowledge base of a **distributed** configuration system is locally incomplete. It is integrated with other configuration systems (e.g. „suppliers’ configuration system) in order to generate consistent product instances for specific customer requirements and many knowledge engineers are maintaining this knowledge base.

Internal vs. external configuration systems. Internal configuration systems are only implemented for a companies internal use. For example, internal configuration systems support sales’ experts in capturing a customer’s requirements and translating them into technical features without errors. However, external configuration systems are designed to provide customers with a direct assistance during product configuration. They are equipped with front-end interfaces to facilitate the configuration task for customers.

Classification of configuration systems according to the interaction nature. Configuration systems can be classified according to the nature of interaction which can be either offline or online. Offline configuration systems work independently from networks. The necessary data for configuration is stored on a data carrier such as an USB-stick, CD-ROM, or DVD-ROM. After product configuration, customers can send the specifications via e.g., e-mail or fax. However, **online** configuration systems enable a communication with customers over the internet. The configuration knowledge is stored on a central web-server. Therefore, the knowledge base can be updated efficiently.

2.1 Basic Techniques for Product Configuration Systems

Online configuration systems can be further divided into two categories: online configuration systems with central data processing and online configuration systems with local data processing. Online configuration systems with local data processing require the load of the configuration application (Java Applets, Full Java Applications) onto the customer's local unit. However, configuration systems with **central data processing** are characterized by continuous communication between the supplier's central server and the customer's local PC, notebook, tablet, or mobile phone.

Classification of configuration systems according to the updates' execution. The updates' execution can be either push or pull. A **push** mode is realized when the supplier's central server containing the product configuration communicates product updates to the customer's local device. In this mode, the central server imposes the updates that have to be accepted by the local device. In contrast, one speaks about a pull mode when the local device retrieves the updates if required.

Classification of configuration systems according to the scope of use. Configuration systems can be categorized as single-purpose and general-purpose systems. A **single-purpose** system is developed to support the sales-delivery process of a product or a set of products of only one company or business field. Single-purpose configuration systems are called special-purpose configuration systems and may be designed for a particular industry such as e.g., bike and car industry. However, general-purpose systems are used to configure diverse product types in different companies (e.g., knowledge-based notebook and mobile phone recommendation systems) [Tiihonen and Soininen, 1997].

Classification of configuration systems according to their complexity. Product configuration systems can be classified according to their design complexity. Tiihonen and Soininen [1997] distinguish between primitive, interactive and automatic systems. Primitive product configuration systems are the simplest ones. They merely record the configuration decisions made by the user without checking the validity of decisions. However, **interactive**

2 Related Work

configuration systems are capable of checking as to whether the conjunction of product knowledge with the customer requirements is valid. They also guide users in making all of the necessary decisions. In addition to the range of functions of interactive configuration systems, automatic ones are able to provide full support and to automatically generate parts or even entire configurations.

Classification of configuration systems according to their integration level.

At the integration level, we can distinguish between stand-alone, data-integrative and application-integrative configuration systems. Stand-alone configuration systems cannot be integrated because they do not dispose of interfaces to other information systems. They just consist of an input for a model, a solver, and an output for the result. **Data-integrative** configuration systems enable one to store the model (see Figure 1.1). However, application-integrative configuration systems enable the integration of whole applications. For example, when configuration systems and CAD-system are integrated, drawings of new parts or components can be automatically generated for customers.

Classification of configuration systems according to the solution searching approach.

There are two main solution-searching approaches: either by technical elements or by features. Searching by technical elements means that the configuration systems enables customers to start from a standard product and then to specify step-by-step product options (defaults; e.g., [Felfernig et al. \[2014c\]](#); [Mandl et al. \[2011a\]](#)).³ However, a configuration system is working by attributes provided to customers with the possibility to specify their requirements. Then, configuration systems search for product variants that best fit to the attributes specified by customers (**features**).

Classification of configuration systems according to their support of the product life cycle.

The product life cycle support refers to product reconfiguration. This is necessary when customers would like to upgrade the

³Critiquing-based recommendation systems work in this way [[Chen and Pu, 2006](#)].

2.2 Types of Product Configuration Systems

product by new or better attributes or to replace non-functioning parts or modules for which identical replacements no longer exist [Sabin and Weigel, 1998]. The different cases that can be encountered are: (a) configuration system without reconfiguration, (b) separate configuration system and reconfiguration system and (c) **integrated configuration system** with reconfiguration.

We use a morphological box to sum up all possible combinations of classifications. The morphological box was introduced for the first time as an efficient tool for creativity and structuring of ideas by Zwicky [1966]. The main advantage of morphological boxes is that they present in a straightforward manner all of the possible solution alternatives for a specific problem. Therefore, we present all of the results of the configuration systems' classification in a morphological box (see Table 2.1). This model should provide software engineers and developers with the main dimensions to be considered when designing a configuration system. The decisions to be made relate essentially to the values to be taken by each dimension. The highlighted fields in Table 2.1 show the relevant characteristics of the examples in this thesis. In the following Section this thesis describes three different types of model-based configuration systems.

2.2 Types of Product Configuration Systems

This section shows, how constraint-based configuration systems, knowledge-based recommendation and MAUT-based systems can be used as product configuration systems. We explain subtypes of those systems and lists preliminaries for each system.

Knowledge base	Rule-based	Case-based	Model-based
Strategy	Assemble-to-order	Fabricate-to-order	Engineer-to-order
Organization	Central	Distributed	
Internal/external	Internal	External	
Interaction nature	Online central	Online local	Offline
Interaction nature	data processing	data processing	
Updates' execution	push	pull	
Scope of use	Single purpose	General purpose	
Complexity	Primitive	Interactive	Automatic
Integration level	Stand-alone	Data-integrative	Application-integrative
Solution searching approach	Technical elements	Features	
Product life cycle support	Conf. system without reconf.	Separate conf. system and reconf. system	Integrated conf. system with reconf.

Table 2.1: Classification of configuration systems. The highlighted cells in this table show the characteristics of the examples in this thesis (see section 3.3).

2 Related Work

Constraint-based Product Configuration⁴

Product configuration systems are important enablers of the mass customization paradigm: people can customize a product according to their own needs [Pine, 1992]. They are considered to be among the most successful applications of artificial intelligence technology.

Product configuration systems can be implemented at the interface between a supplier and its customers over the Internet in order to support the configuration task. Given a set of customer requirements and a product family description, the configuration task is to find a valid and completely specified product structure among all alternatives that the generic structure describes [Sabin and Weigel, 1998]. In this context, customers are provided with the possibility to alter a basic product and to graphically visualize the effects of these changes.

Several techniques exist to model product configuration knowledge in a constraint-based system. In the following, we give a short introduction into some of those systems.

Static CSP (SCSP). A static constraint satisfaction problem (CSP) is a tuple (V, D, C) where V is a set of all variables. Those variables model product attributes and customer requirements. The set D consists of all domain elements for each variable, s.t., $D = \{dom(v_1) = \{dom_1, \dots, dom_n\}, \dots, dom(v_n) = \{dom_1, \dots, dom_n\}\}$. A set of constraints C restricts the number of consistent assignments for the variables. This technique is used in this thesis and therefore explained in detail in Section 3.1.

⁴This section is taken from Hotz et al. [2014].

2 Related Work

Satisfiability problems (SAT). A SAT problem is similar to static CSPs except the variable domains are restricted to the Boolean values *true* and *false*. It is to identify an assignment to the given Boolean variables in such a way that the mentioned formula in conjunctive normal form evaluates to true. This is the case if each clause has at least one literal that evaluates to true. For a more detailed discussion of SAT solving and related algorithms, we refer the reader to [Claessen et al. \[2008\]](#).

Dynamic constraint satisfaction (DCSP). Configuration tasks very often include variables that do not have to be taken into account in specific configuration contexts. For instance, in a notebook configuration, the variable *BlueTooth* is not relevant if the user is not interested in this technology. The property of a new variable *irrelevance* is implemented in terms of variable deactivation; that is, if *BlueTooth* is not part of the configuration, the value of the variable *irrelevance* is set to 0 (*false*). Such a reasoning about activity states in DCSPs is based on predefined activity constraints. A formal discussion of the properties of DCSPs can be found in [Bowen and Bahler \[1991\]](#). The concept of variable activation has also been applied in other approaches. For example, generative constraint satisfaction (GCSP) based systems [[Stumptner et al., 1998](#)] generalize the concept of variable activation to the concept of component and constraint activation. The configuration environment of SIEMENS is based on such a generative approach [[Falkner and Schreiner, 2014](#)].

Generative constraint satisfaction (GCSP). Major drawbacks of static (and dynamic) CSPs include representational limits in terms of dealing with variables and variable values instead of component types and corresponding components (instances). Another disadvantage is that static CSPs are limited with regard to the description of configuration problems in which the size and structure of configurations cannot be estimated beforehand. Generative constraint satisfaction (GCSP) helps to overcome these problems by moving variables and corresponding constraints to the level of component types and meta-constraints (generic constraints). Generic constraints (constraint schemata) help to identify and generate relevant additional components and

2.2 Types of Product Configuration Systems

Criteria	SCSP	SAT	DCSP	GCSP
Standard graphical modeling concepts?	-	-	-	-
Component-oriented modeling	-	-	-	+
Automated consistency maintenance	~	~	~	~
Modularization concepts available	-	-	-	+
Support of easy knowledge base evolution and maintenance	~	~	~	~
Model-based knowledge representation	+	+	+	+
Efficient reasoning	+	+	+	+
Able to solve generative problem settings	-	-	-	+
Able to provide explanations	+	~	+	+

Table 2.2: Evaluation of constraint representations [Hotz et al., 2014]. + = good support, ~ = some support, - = no support

variables that are relevant in the current configuration context. A detailed description of GCSPs can be found in [Stumptner et al. \[1998\]](#).

We now compare the configuration knowledge representations that have been discussed within the scope of this section (SCSP = static constraint satisfaction, SAT = SAT solving, DCSP = dynamic constraint satisfaction, GCSP = generative constraint satisfaction). We provide an overview based on nine criteria in Table 2.2. A detailed description of the criteria and an overview of other representation techniques is given in [Hotz et al. \[2014\]](#).

This thesis shows how we can reduce the effort for maintaining constraint-based product configuration systems via detecting anomalies (Section 4.1), explaining the anomalies (Section 4.2), generating recommended items from the knowledge base to a particular knowledge engineer (Section 5.1), and calculate metrics to evaluate the difficulty for maintenance tasks (Section 5.4). Examples 3.2 and 3.3 represent constraint-based product configuration systems.

Knowledge-based Product Recommendation⁵

The main difference between constraint-based product configuration and knowledge-based product recommendation is, that product configuration systems have one configurable product (e.g., the brand *Golf* of the manufacturer *Volkswagen*) whereas knowledge-based product recommendation present a set of predefined products to a customer and she has to decide, which product fits her needs.

Compared to other recommendation systems knowledge-based product recommendation systems are characterized by the two knowledge aspects not found in the other designs: namely user requirements and domain knowledge. Obviously, there are collaborative and content-based recommendation systems that allow users to pose queries or that have some forms of heuristics with respect to their content. What distinguishes a knowledge-based approach is that its emphasis: emphasis on the user's situation and how recommended items can meet that particular need.

If we consider the knowledge sources used in collaborative and content-based recommendation, it is clear that the knowledge-based category itself is something of an accident of history. Systems that used additional knowledge sources came to be defined as "knowledge-based" because they relied more heavily on knowledge sources that were not being employed by the more widely-used techniques.

There are two well-known approaches to knowledge-based recommendation: case-based recommendation [Burke, 2000; Mirzadeh et al., 2005; Ricci and Nguyen, 2007; Smyth et al., 2004] and constraint-based recommendation [Thompson et al., 2004; Felfernig et al., 2007b]. In terms of used knowledge sources, both approaches are quite similar. Systems of both types must, for example, collect the requirements of the current user in order to derive new

⁵This section is taken from Felfernig and Burke [2008].

2.2 Types of Product Configuration Systems

solutions, propose repairs in situations where no solution could be found and support explanations for the recommended items.

Case-based recommendation treats recommendation as primarily a similarity-assessment problem. How can the system find a product that is most similar to what the user has in mind, with the understanding that what counts as similar will often involve domain-specific knowledge and considerations. **Constraint-based recommendation** takes into account explicitly defined constraints (e.g., filter constraints or incompatibility constraints). If no item really fits the wishes of a customer (the calculated similarity value exceeds a certain threshold for all relevant products or the set of constraints is inconsistent with the given set of customer requirements) both knowledge-based approaches exploit mechanisms supporting the determination of minimal set of changes to the given set of customer requirements such that a solution can be found - see, for example [McSherry \[2004\]](#); [Felfernig et al. \[2007b\]](#).

The interaction with a knowledge-based recommendation application is typically modeled in the form of a dialog (conversational recommender) where users can specify their requirements in the form of answers to questions or rates for product attributes [[Burke, 2000](#); [Thompson et al., 2004](#); [Mirzadeh et al., 2005](#); [Ricci and Nguyen, 2007](#); [Felfernig et al., 2007b](#)]. This dialog can be modeled explicitly, for example, in the form of a finite state automaton (see, e.g., [Felfernig et al. \[2007b\]](#)) or in a way which allows the user to select interesting attributes (questions) on her own [[Mirzadeh et al., 2005](#)]. Furthermore, user interaction with a recommender application can be enriched with natural language interaction [[Thompson et al., 2004](#)] which allows a more flexible interaction processes. For example, users can specify component properties on a textual level without being forced to answer a potentially larger number of questions. Another interesting aspect of additional textual interfaces is the flexibility to support queries which are not directly related to product search but to issues such as questions regarding the functionality or technical questions.

2 Related Work

Many case-based recommendation applications support the concept of critiquing (see, e.g., [Burke \[2000\]](#); [Smyth et al. \[2004\]](#); [Chen and Pu \[2006\]](#)), in which the user responds to a recommended item by identifying how it differs from their ideal. For example, a user presented with a restaurant featuring a traditional style of food may apply the critique "More Creative" and obtain a more contemporary take on the same cuisine [[Burke et al., 1997](#)]. Such an interface has the advantage of allowing the user to formulate requirements on the fly, in response to examples. Critiquing interfaces promise a faster identification of interesting recommendations in terms of less decision effort, better decision accuracy, and increase of a user's confidence in a decision [[Chen and Pu, 2006](#)]. Recent research has moved from unit critiques, in which the user identifies specific item properties to critique (e.g., "I would prefer a camera with a lower price"), to more complex compound critiques, that move along several feature dimensions at once. Such critiques can be manually engineered (as in [Burke et al. \[1997\]](#)) or can be automatically generated from the existing product assortment by mining association rules representing representative critique patterns in the available assortment [[Smyth et al., 2004](#)].

This thesis uses the knowledge-based product recommendation technique for explaining assignment-based redundancy detection (Section [5.2.2](#)), detecting dependencies between items in the knowledge base (Section [5.3](#)), and generating test cases (Section [5.5](#)). Furthermore, the study in Section [4.4](#) is based on a knowledge-based product recommendation. Example [3.1](#) describes a knowledge-based product recommendation.

Multi-Attribute Utility Theory⁶

Products included in a recommendation have to be ranked according to their relevance for the customer [[Felfernig et al., 2006a](#); [Felfernig and Burke, 2008](#)]. In the line of serial position effects which induce customers to

⁶This section is taken from [Felfernig et al. \[2013c\]](#).

2.2 Types of Product Configuration Systems

preferably take a look at and select items at the beginning of a list, the high-ranking of the most relevant items is extremely important [Gershberg and Shimamura, 1994; Lashley, 1951]. For the determination of such rankings we apply the concepts of Multi-Attribute Utility Theory (MAUT) [Keeney and Raiffa, 1976; Schmitt et al., 2003; von Winterfeldt and Edwards, 1986] where each product is evaluated according to a predefined set of interest dimensions which are abstract evaluation criteria for products. Profit and availability are examples for such interest dimensions in the domain of financial services. For example, if a customer is interested in high return rates and long term investments, the dimension profit is very important. Consequently, customer requirements influence the importance of corresponding interest dimensions.

Basically, MAUT-based systems describe products in terms of n product variables v_1, \dots, v_n which can assume values from the domains $dom(v_1), \dots, dom(v_n)$. A concrete product c_p can be represented as a tuple $c_p = \langle dom_1 \in dom(v_1), \dots, dom_n \in dom(v_n) \rangle$. The overall utility for a product c_p is defined by

$$U(c_p) = \sum_{i=1}^n \omega_i \times u_i(dom_i \in dom(v_i)) \text{ with } \sum_{i=1}^n \omega_i = 1 \quad (2.1)$$

where dom_i is c_p 's concrete value for the i th variable v_i , ω_i is the importance weight of v_i as compared to the other attributes, and u_i is a value function representing the respective utility values of the various possible instantiations of v_i . That is, while the various ω_i quantify the impact of a variable on the user's overall evaluation of an object, the functions u_i represent the user's preferences regarding instantiations of product variables [Schmitt et al., 2003].

We use MAUT to describe techniques to repair knowledge bases (Section 4.3) and to evaluate the presentation of products which do not fulfill all of the customers' requirements compared to the presentation of conflicts and diagnoses in Section 4.4. Example 3.4 represents a MAUT system.

2.3 Maintaining Product Configuration Systems

This section gives an overview of the state-of-the-art of several techniques this thesis uses to create / enhance supporting tools for knowledge engineers and end-users of product configuration systems.

Recommendation Techniques⁷

A recommendation technique is a set of knowledge sources and an algorithmic approach to generate recommendations using those sources. There are of course no a priori limitations on how these different sources of knowledge can be combined, and in fact, the field of recommender systems has seen great diversity in approaches. However, certain approaches have turned out to be practical and effective and are generally accepted techniques. The following discussion is a brief overview of three common types of recommendation techniques: collaborative filtering, content-based recommendation, and knowledge-based recommendation.

Collaborative Filtering

The approach most closely associated with recommender systems since the field's inception is of course collaborative filtering or collaborative recommendation.⁸ The knowledge sources here are collaborative opinion profiles, demographic profiles, and user opinions. Of course one of the strengths of this technique is that little else is required to implement it in its pure form.

⁷This section is mainly taken from [Felfernig et al. \[2013b\]](#) and [Felfernig and Burke \[2008\]](#).

⁸The 'filtering' terminology is a legacy from an early application: filtering of interesting Usenet messages [[Konstan et al., 1997](#)].

2.3 Maintaining Product Configuration Systems

Many algorithmic approaches have been applied to these knowledge sources, but in general, the problem can be seen as a form of multi-way classification task. It differs from classic classification tasks in that there is no specific dependent class variable being predicted in all situations, but rather the system may be called upon to make predictions about any item for any user. Thus collaborative recommendation is not amenable to a strict application of the tools of machine learning.

The most well-known approach is that of a nearest neighbor, in which ratings are extrapolated by comparing the user opinion knowledge against collaborative opinions and extrapolations made [Resnick et al., 1994]. The popular item-based variant treats the collaborative information as features associated with items rather than with users [Sarwar et al., 2001]. Model-based techniques have been employed to compress the collaborative opinion data, including clustering, singular value decomposition, and others [Sarwar et al., 2001].

What all of these methods share is their sole reliance on opinions or rating data as the knowledge source for recommendations. There are well-known characteristics that result from this choice. First, there is positive benefit that no other information about users or items is required. This makes the approach attractive for hard to characterize items, like movies and music.

This technique can be said to be the most mature of the recommendation technologies and the characteristics of these algorithms are well established. Collaborative recommendation works best when there is a large amount of collaborative knowledge available and when there is a substantial history of user opinions. The density of the collaborative user-item matrix is a substantial consideration.

2 Related Work

Collaborative recommendation can theoretically be applied in any domain. It places very little restriction on the types of items that can be recommended and is highly suitable for items in which user tastes vary for reasons that might be difficult to represent explicitly, such as music and movies. However, there are considerations that may make a collaborative approach less attractive. One is opinion density. A user can listen to dozens of music tracks in a day, and might watch dozens of movies a year, but she would unlikely to live in dozens of apartments, own dozens of cars or have dozens of different pets. In some areas it is simply more difficult to accumulate a pattern of preference. It is possible to ask the user to venture a prospective opinion without having actually experienced the item in question, but such opinions will generally be quite speculative compared to those arrived at through experience.

We use collaborative filtering to distinguish relevant information in a knowledge base from unnecessary information related to a maintenance task. The term 'information' subsumes all objects in a knowledge base like constraints and variables. A detailed description of this technique in the context of supporting tools for knowledge engineers is given in Section 5.1.

Content-based Recommendation

Content-based recommendation [Pazzani and Billsus, 1997] is more like a pure classification task in the machine learning sense. The task is to learn a specific classification rule for each user on the basis of the user's rating information and the attributes of each item so that items can be classified as likely to be interesting or not. No social knowledge is used. The content-based problem has been tackled using a variety of machine learning techniques. However, in general, some of the more sophisticated techniques have been less successful, due to the sparsity problem. An individual user profile may not, in many cases, have enough data for a reliable profile. Simple techniques such as k-nearest neighbors and naive Bayes have often proved effective here.

2.3 Maintaining Product Configuration Systems

Because a content-based recommender has access to item features (e.g., keywords or categories), it does not suffer from the new item problem: new items look just like old items. The new user problem remains since users must build up a sufficiently rich profile through the addition of multiple ratings. Depending on the feature set and learning algorithm, however, a small number of ratings may be sufficient.

Indeed, the quality of the data set becomes of primary importance in a content-based system. The learned profiles of each user will only be as good as the system's level of detail in representing the distinctions that matter in the domain. The creators of the music recommender Pandora (www.pandora.com) first developed a highly-detailed representation of musical form and expression before attempting to build a recommendation system. Developers must make sure that all items in the catalog have a uniform, detailed and complete representation: a combination which can be difficult to achieve in many e-commerce contexts.

Based on variables and domain elements a constraint has, we can use this technique to find similar constraints. This technique is denoted as 'dependencies between constraints' and its practical use in the context of supporting knowledge engineers will be described in Sections 5.1 and 6.2.

Knowledge-based Recommendation

This thesis described maintenance techniques for knowledge-based product recommendation systems in the previous Section. We refer the reader to that section and Mandl et al. [2011a]; Jannach et al. [2010]; Felfernig et al. [2008a, 2007a, 2006a,c]; Burke [2000].

Anomaly Management⁹

Anomalies can be characterized as parts of a knowledge base that conform to a defined *pattern of unintended structures* [Chandola et al., 2009]. Current research is focusing on *automated testing and debugging* of configuration knowledge bases [Friedrich et al., 2014]. Anomaly management operations are very important since configuration knowledge bases are often complex and subject to frequent changes [Fleischanderl et al., 1998; Barker et al., 1989].

The foundations for anomaly management are *conflict detection* and *diagnosis* algorithms that help to detect a) *minimal subsets* of constraints of the knowledge base that are responsible for a faulty behavior [Felfernig et al., 2013c, 2012b, 2004a; Junker, 2004] and b) *minimal subsets* of constraints which resolves all conflicts in the knowledge base [Felfernig et al., 2004a; Felfernig and Schubert, 2011b; Felfernig et al., 2013d]. More precisely, conflict detection algorithms are able to determine minimal sets of constraints that are inconsistent; that is, do not allow the determination of a solution (valid configuration respectively no products can be found). In addition, diagnosis algorithms can determine minimal sets of constraints in the configuration knowledge base that have to be deleted or adapted such that the remaining set of constraints is consistent [Friedrich et al., 2014]; that is, a solution can be determined. Typically, diagnoses are determined from a given set of (minimal) conflicts and, vice versa, minimal conflicts can be determined on the basis of a given set of (minimal) diagnoses.

While conflict detection (e.g. *QUICKXPLAIN*) and diagnoses (e.g. *FAST-DIAG*, see Section 4.1.1) algorithms are well discussed, this thesis extends the work of anomaly management. On the one hand this thesis describes new types of *anomalies* - redundancies (see Section 4.1.2) and well-formedness violations (see Section 4.1.3) - and, on the other hand, extends the term *management* with explanations (see Section 4.2) of these anomalies, repair

⁹This section is based on Felfernig et al. [2013c] and Felfernig et al. [2014e].

2.3 Maintaining Product Configuration Systems

of scoring rules (see Section 4.3), and the visualization of inconsistencies to end-users (see Section 4.4).

Simulation

A simulation model is a computerized model that represents some dynamic system or phenomenon. One of the main motivations for developing a simulation model or using any other modeling method is that it is an inexpensive way to gain important insights when the costs, risks, or logistics of manipulating the real system of interest are prohibitive. Simulation can be used to deal with uncertainty (e.g., unknown dependencies between elements in a configuration knowledge base), the dynamics of configuration models (e.g., changes in a configuration knowledge base), and to have feedback on interacting with the model (e.g., getting information about effects when the knowledge base is changed [Kellner et al., 1999]).¹⁰ At the moment, no research has been done in the context of using simulation techniques concerning developing and maintaining configuration knowledge bases. Next, we give a short overview of usage scenarios for simulation in related research areas and similar application scenarios.

Simulation for Requirements Engineering. Complex product configurations are mainly used iteratively, s.t. the requirements are not available at once. To generate a valid product Rabiser and Dhungana [2007] use the simulation technology to complete a configuration. The simulator can be set up using a default configuration for not yet taken decisions to allow its use even if only few decisions have been taken. Feedback to the simulated configuration help to a) get new requirements and b) get a deep understanding of the requirements [Rabiser and Dhungana, 2007].

¹⁰A detailed description of the advantages of simulation is given in [Kellner et al., 1999].

2 Related Work

Simulation for production. In production management many decisions have to be taken under uncertainty. The simulation technique can help to reduce the uncertainty via planning product lines and logistics forecasting. For example, the use of Monte Carlo simulations can predict the number of produced goods and optimal buffers can be calculated to have an optimal scalability with low costs for the buffers in mass customization manufacturing processes [Daaboul et al., 2009; Rabe and Jäkel, 2002; Fisher and Ittner, 1999].¹¹

In the context of configuration knowledge base development and maintenance we can adapt the application scenarios to approximate several metrics, rank diagnoses (see Section 5.4), and approximate the possibility that a combination of variable assignments is consistent (see Section 6.2).

Evaluation and Verification

Evaluation is the verification and validation of constraint-based systems [Preece, 1998]. Verification is the process of checking whether the software system meets the specified requirements of users, while validation is the process of checking whether the software system meets the current requirements of the users [Boehm, 1984]. A positively verified system has been built in conformance with a number of well-defined properties, chiefly freedom from logical conflict, redundancy, and deficiency [Preece et al., 1992].

O’Keefe and O’Leary [1993] give an overview of the state-of-the-art for the evaluation of knowledge bases (see Table 2.3).¹² Validators typically use methods to investigate the components of a KBS (individual rules or constraints, programmed heuristics, the knowledge or conceptual model), and the entire KBS itself [O’Keefe and O’Leary, 1993].

¹¹For a detailed overview of simulation in production we refer the reader to Olhager and Persson [2007].

¹²Note that these authors are focusing on rule-based systems.

2.3 Maintaining Product Configuration Systems

General approach	Specific approach	timing in process
Component	Rule validation - manually investigate important rules	early
	Heuristic - compare performance to optimum - worst case analysis - distribution sampling	middle
	Meta-Models - construct and maintain conceptual model of the knowledge base	all
System	Case testing - KBS solves cases, performance compared to expert	all
	Turing test - KBS and experts solve cases, performances evaluated by third-party	late
	Simulation - KBS controls simulation model, performance evaluated	late
	Control group - KBS implemented for test group, performance compared to control group	very late
	Sensitivity analysis - cases altered and input / output relationships evaluated	all
	Other model - another model, e.g. quantitative or induced - Constructed and performance compared to KBS	late
	Line of reasoning - line of reasoning on test cases compared to elicitation material or expert	middle

Table 2.3: Methods for KBS validation and timing in the life-cycle [O'Keefe and O'Leary, 1993]

2 Related Work

A typical method for the verification of a KBS is the creation of test cases. Knowledge engineers have to create and verify such test cases and a correct KBS returns the expected correct result. There are many difficulties when using this approach for testing a KBS. On the one hand, the knowledge engineers have to know the correct behavior of the KBS. In complex domains this prerequisite could be problematic when even the stakeholders do not know the correct behavior of the KBS. On the other hand, the effort for designing, implementing, and executing all test cases could be very time-consuming and error-prone [Keefe and Preece, 1996].

Preece [1998] defined nine evaluation methods which have to be observed when engineers want to evaluate knowledge bases successfully. Those artifacts can be observed in different phases of a knowledge base development process:

- Requirements specification: typically in natural language written set of desired user requirements
- Conceptual model: describes the knowledge content of the KBS in terms of real-world entities and relations
- Design model: *operationalizes* the conceptual model into an executable KBS
- Implemented system: this is the final product of the development process: the KBS itself
- Inspection: human proof-reading is the most commonly-employed technique
- Static verification: checking the knowledge base for anomalies like conflicts and redundancies
- Formal proof: proof techniques can be employed to verify that the formal artifacts meet the specified requirements
- Cross-reference verification: performs cross-checking between different 'levels' of KBS descriptions (conceptual model and design model and design model and implemented system)
- Empirical Testing: running the system with test cases [Preece, 1998]

2.3 Maintaining Product Configuration Systems

Table 2.4 gives an overview which evaluation method can be used with which artifact.

Artifact	Evaluation techniques
Conceptual model	Inspection, static verification (if formalised), cross-ref verification (against design model)
Design model	Inspection, static verification, formal proof, cross-ref verification (against conceptual model, implemented system)
Implemented system	Inspection, static verification, testing, cross-ref verification (against design model)

Table 2.4: Applicability of evaluation techniques to KBS development artifacts [Prerau, 1987]

To sum up, there exist some overviews, for example, of rule-based systems (see e.g., Keefe and Preece [1996]) but up to now, there does not exist an overview of how to measure the quality of product configuration systems. Based on the previous research for quality measurement in rule-based systems this thesis gives an overview of quality measurement for product configuration systems in Section 5.4.

3 Constraint-based Product Configuration Systems

This chapter gives an overview of constraint-based systems. Section 3.1 introduces the concept of constraint satisfaction problems and Section 3.2 describes basic definitions for those systems. Section 3.3 lists running examples for this thesis.

3.1 Constraint Satisfaction Problems¹

We use constraint satisfaction problems (CSP; Tsang [1993]) to represent constraint-based systems. A constraint satisfaction problem is defined as a triple $KB = \{V, D, C\}$. The following section describes the sets variables V , domains D , and constraints C .

Variables

V is a set of product and customer variables. Those variables can be based on objective observations like the screen size of a mobile phone or the capacity of a memory in a notebook. If product attributes can not be described quantitatively, domain experts can try to *abstract* the characteristics of a product from incomparable values into quantitative values. For example, it is difficult to compare the uncertainty of financial products. Instead,

¹This section is based on Reinfrank et al. [2015a].

3 Constraint-based Product Configuration Systems

domain experts can abstract the uncertainty of a financial product e.g., on a 10-point scale.

This abstraction can also be used, if users of configuration systems or knowledge-based recommendation systems can not compare the value of a product variable. For example, if a user wants to have a notebook for writing texts, reading mails, and surfing in the internet, she possibly does not know the differences of several graphic cards or the characteristics of a card, which are necessary to fulfill her requirements. In such scenarios, the system can replace the quantitative description of graphic cards with subjective descriptions like a usage scenario for notebooks.

All variables have a **selection strategy** $v_{sel} = \{singleselect, multipleAND, multipleOR\}$ which describes if a variable v can have more than one value. $v_{sel} = singleselect$ shows that the variable v can have either zero or one assignment. For example, the product variable $price$ has one assignment (e.g., $price = 399$). If a variable can have more than one value, we differ between $multipleAND$ and $multipleOR$. $v_{sel} = multipleAND$ points out that a variable can have more than one assignment. For example, a notebook can have two wireless connections like $bluetooth$ AND $WLAN$, s.t. $wireless_connection_{sel} = multipleAND$. On the other hand, a customer wants to have a notebook with two OR four cpu_cores . We denote such a selection strategy as $multipleOR$, s.t. $cpu_cores_{sel} = multipleOR$. How the selection strategy can be considered in a configuration knowledge base will be described in the following.

Domains

Each variable $v_i \in V$ has a domain $dom(v_i) \in D$ that contains a set of all possible values (not only the assigned values). The domains can have the following sets of domains:

3.1 Constraint Satisfaction Problems

- **boolean domain:** the domain has two elements. For example, a car can have either automatic or manual gear shift. In feature models [Benavides et al., 2010] each variable must have a domain with the values $\{true, false\}$.
- **finite elements:** the common type of domains for knowledge-based recommendation contains a restricted number of elements. For example, the brands for a bike are restricted to a finite number.
- **interval:** if a user of a configuration system has to insert the price preferences, the system may offer intervals as a representation for the price preferences.
- **infinite elements:** theoretically, a variable (e.g., size) can be infinite. Such domains do not exist in product configuration systems in practice. Hence, this thesis focuses on boolean and finite domains.

Another characteristic of domain elements is **ordering**. With respect to relations between variables and domain elements we first have to define the ordering of the domain elements. Boolean domains and textual domains usually can not be ordered. In such cases we can reduce the number of available relations to $REL = \{=, \neq\}$. On the other hand, numeric domain elements can be ordered and we extend the set of relations to $REL' = \{=, \neq, <, \leq, \geq, >\}$.

Finally, we also have to define which of the values within the domain should be **preferred**. For example, we assume that a low weight of a bike and a long guarantee period will be preferred. McSherry [2003] denotes such situations as *lower is better* (*LIB*, e.g., for the weight) and *more is better* (*MIB*, e.g., for the guarantee). A third situation is constituted, if we can not define whether more or less is better. For example, a high number of gears is necessary for mountain biking whereas a low number of gears is better for street racing. Such situations are denoted as *nearer is better* (*NIB*; McSherry [2003]).

3 Constraint-based Product Configuration Systems

Constraints

C describes a set of constraints. A constraint describes a relation between one or more variables and assignments. For example, constraints define relationships between product variables (e.g., $price \leq 300$ implies $frame_material = aluminium$) or between product and customer variables (e.g., $customer_size \geq 190cm$ implies $frame_size \geq 60cm$). Those constraints can be divided into different subsets of constraints which are described in the following.

Some types of constraint-based systems do not have an open space for solutions but have *products*. For example, knowledge-based recommendation systems are a subset of constraint-based systems and are typically based on a set of predefined products. For example, mobile phones are typically predefined and their components can not be changed. Within a constraint-based system a product can be presented as a conjunctive constraint with all product variables. The set of all product constraints is a disjunctive query, s.t., $C_P = \{product_0 \vee product_1 \vee \dots \vee product_n\}$.

The set C_P can be used to differ between product configuration and knowledge-based recommendation. *Product configuration* can be described as knowledge-based configuration with predefined variables and domains but it does not contain a specific set of predefined products, s.t., $C_P = \emptyset$. Each possible combination of product variable assignments is a valid configuration unless the variable assignments and their combination is not in conflict with C_{KB} . For example, the car manufacturer BMW offers 10^{17} different consistent configurations for a BMW series 7 [Hu et al., 2008]. On the other hand, *knowledge-based recommendation* contains a set of products. The products are defined in C_P and $C_P \neq \emptyset$. Typically, mobile phones can not be configured, but potential customers can select a mobile phone from a huge assortment based on product attributes (e.g., screen size).

3.1 Constraint Satisfaction Problems

Customer requirements represent the preferences of customers in the recommendation / configuration process. The set of customer preferences is denoted as C_R . For example, a customer can have the preference that a bike should be cheaper than 599 EUR, s.t., $\{price < 599\} \in C_R$.

Knowledge base constraints - which are denoted as filters in feature models - in C_{KB} define the relationship between variables and are defined in the set C_{KB} . For example, the relationship between the customers' *Usage* and the product attributes is $c_1 := Usage = Competition \rightarrow BikeType = RacerBike \wedge FrameSize = 60cm$; The aggregation of customer requirements, constraints, and products represent the constraints, s.t. $C_R \cup C_{KB} \cup C_P = C$.

Each constraint c can be divided into **assignments**. The set of assignments within a constraint is denoted as $A(c)$. An assignment $a \in A(c)$ consists of one variable v , one relationship $rel \in REL$ respectively $rel \in REL'$, and one value d which is an element of the domain $dom(v)$. The different types of available relationships depend on the values in the corresponding domain.

To consider the selection strategy of variables within the constraints, we have to duplicate the variables. For example, if a customer wants to have a bike for *Competition* and *everydayLife*, we replace the variable $Usage \in V$ by $Usage1 \in V$ and $Usage2 \in V$ and the domain will also be duplicated, s.t., $dom(Usage1) = dom(Usage2) = dom(Usage)$. We also have to extend the affected constraints in C , such that we have to replace the affected assignments in the example constraint $Usage = Competition \rightarrow BikeType = RacerBike \wedge FrameSize = 60cm; \in C_{KB}$ by $(Usage1 = Competition \vee Usage2 = Competition) \rightarrow BikeType = RacerBike \wedge FrameSize = 60cm$; and the customer requirements $Usage1 = Competition \wedge Usage2 = everydayLife$ for the selection strategy $Usage_{sel} = multipleAND$.

MAUT-based systems need further sets of constraints. The following list gives an overview of the required sets for MAUT-based systems.

3 Constraint-based Product Configuration Systems

- Customer requirements (C_R) define the requirements of a *concrete* customer, for example, a medium term investment period and money for rainy days (see Table 3.4).
- Constraints (C_{KB}) define, for example, which products should be recommended and which restrictions on combinations of requirements exist (see Table 3.13).
- A product catalog (C_P) defines the available product assortment (see Table 3.8).
- Scoring rules (UC) determine a ranking in which items (products and services) of a recommendation result are presented to the customer (see Tables 3.2, 3.3, 3.9, 3.10).
- Repair constraints (R) expand a value for a product or customer variable v .
- Example constraints (E) are, e.g., provided by the marketing department and represent a correct behavior of the recommendation system (see Table 3.16).
- Summing constraints (S) aggregate the customer utilities (see Table 3.14).

This section described CSPs for constraint-based configuration, knowledge-based recommendation systems, and MAUT-based systems. The next section describes consistent and inconsistent knowledge bases and detects several types of anomalies in constraint-based systems.

3.2 Definitions²

This section defines basic terms in the context of constraint-based configuration systems.

²This section is based on Reinfrank et al. [2015a].

3.2 Definitions

The constraint set C restricts the set of valid instances. While C_{KB} and C_P remain stable during a user session, a potential customer adds her preferences in the set C_R . An instance is given if at least one customer preference is added to C_R . Definition 1 introduces the term 'instance'.

Definition 1 'Instance': *An instance is given if at least one constraint is in the set C_R , s.t. $C_R \neq \emptyset$.*

In a complete instance all variables in the knowledge base have at least one assignment. Definition 2 introduces the definition for a complete instance.

Definition 2 'Complete Instance': *An instance is complete iff all variables have an assignment, such that $\forall v \in V v$ is assigned.*

Instances can either fulfill all constraints in the constraint set C (consistent) or not (inconsistent). Definition 3 defines the term 'consistent instance'.

Definition 3 'Consistent Instance': *An instance (complete or incomplete) is consistent, if no constraint in C is violated.*

Constraint-based systems can have some anomalies in terms of conflicts, redundancies and well-formedness violations. Figure 3.1 gives an overview of different types of anomalies. In the following, this thesis defines the anomalies.

3 Constraint-based Product Configuration Systems

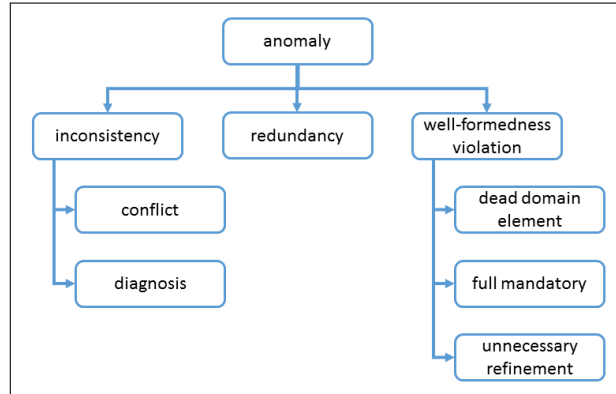


Figure 3.1: Different types of anomalies.

In constraint-based systems it can happen that the system can not offer consistent instances to a user (**inconsistency**) because it is not possible to satisfy all constraints (see Definition 3). Such a 'no solution could be found' dilemma is caused by at least one conflict between a) the constraints in the knowledge base C_{KB} and the products C_P on the one hand and the customer requirements C_R on the other hand (customer requirements can not be fulfilled) or b) within the sets C_P and C_{KB} (knowledge base development and maintenance problem). Definition 4 introduces a formal representation of a conflict.

Definition 4 'Conflict': A conflict is a set of constraints $CS \subseteq C$ which can not be fulfilled by the CSP, s.t. CS is inconsistent.

If we have an inconsistency in our knowledge base, we can say that C is always a conflict set. To have a more detailed information about the inconsistency, we introduce the term 'minimal conflict' which is described in Definition 5.

Definition 5 'Minimal Conflict': A minimal conflict CS is a conflict (see Definition 4) and the set CS only contains constraints which are responsible for the conflict, s.t. $\nexists_{c \in CS} CS \setminus \{c\}$ is inconsistent.

3.2 Definitions

A knowledge base can have more than one conflict. In such cases we can help users to resolve all conflicts with diagnoses. A diagnosis Δ is a set of constraints. The removal of the set Δ from C leads to a consistent knowledge base, formally described in Definition 6.

Definition 6 'Diagnosis': A diagnosis Δ is a set of constraints $\Delta \subseteq C$. When removing the set Δ from C , the knowledge base will be consistent, s.t. $C \setminus \Delta$ is consistent.

Assuming that C_{KB} is consistent (see Definition 3) and C_P contains zero (product configuration) or at least one product (knowledge-based recommendation), we can say that the knowledge base always will be consistent if we remove C_R . In Definition 7 we introduce the term 'minimal diagnosis' which helps to reduce the number of constraints within a diagnosis without losing the property of being a diagnosis.

Definition 7 'Minimal Diagnosis': A minimal diagnosis Δ is a diagnosis (see Definition 6) and there doesn't exist a subset $\Delta' \subset \Delta$ which has the same property of being a diagnosis.

After having calculated at least one diagnosis and removed all constraints from the knowledge base which are in one diagnosis, we can ensure a consistent knowledge base that is necessary for calculating redundancies and well-formedness violations.

A redundancy is a set of redundant constraints in the knowledge base. A constraint c is redundant if a knowledge base KB' without the constraint c has the same semantics as the knowledge base KB which contains the constraint. We use the term *semantics* to describe a knowledge base KB' with the same solution set as KB . Redundant constraints are formally described in Definition 8.

3 Constraint-based Product Configuration Systems

Definition 8 'Redundant constraint': A constraint c is redundant iff the removal of the constraint from C_{KB} leads to the same semantics, s.t. $C_{KB} \setminus \{c\} \models c$.

While conflicts, diagnoses, and redundancies focus on constraints, well-formedness violations identify anomalies based on variables and domain elements [Felfernig et al., 2013a]. We now introduce well-formedness violations in product configuration systems.

The first well-formedness violation focuses on *dead domain elements*. A dead domain element is an element which can never be assigned to its variable in a consistent instance (see Definition 3). Definition 9 introduces a formal description of dead elements.

Definition 9 'Dead domain elements': A domain element $val \in dom(v)$ is dead iff it is never in a consistent instance, s.t. $C \cup \{v = val\}$ is inconsistent.

On the other hand, we can have domain elements which are assigned to each consistent instance. We denote such domain elements *full mandatory* and therefore we introduce Definition 10.

Definition 10 'Full mandatory': A domain element $val \in dom(v)$ is full mandatory iff there is no consistent (complete or incomplete) instance where the variable v does not have the assignment val , s.t. $C \cup \{v \neq val\}$ is inconsistent.

Another well-formedness violation is called *unnecessary refinement*. Such an unnecessary refinement consists of two variables. If the first variable has an assignment, it is possible to predict the assignment of the second variable because the second variable can only have exactly one consistent assignment. A formal definition is given in Definition 11.

Definition 11 *'Unnecessary refinement'*: A knowledge base contains a variable pair v_i, v_j . For each domain element val_1 of variable v_i , we can say that variable v_j always has the same assignment $v_j = val_2$, s.t. $\forall_{val_1 \in dom(v_i)} \exists_{val_2 \in dom(v_j)} C \cup \{v_i = val_1 \wedge v_j \neq val_2\}$ is inconsistent.

To repair MAUT-based recommendation systems, we define a repair task as constraint set adaptation problem, which is described in Definition 12.

Definition 12 *'Constraint Set Adaptation Problem'*. A constraint set adaptation problem is defined by the tuple (V, D, UC, Con, Opt) , where V is a set of variables referred to by the constraints in $Con = R \cup S \cup E$. D contains the domain definitions of the variables in V and UC represents the set of scoring roles inconsistent with the examples in E . Opt is the optimization function of the underlying nonlinear optimization problem.

If a set of new assignments for all variables in a MAUT-based recommendation problem is found - according to the constraints in C and examples in E - we have a valid constraint set adaptation. Definition 13 describes a constraint set adaptation.

Definition 13 *'Constraint Set Adaptation'*. A constraint set adaptation for a given constraint set adaptation problem (V, D, UC, Con, Opt) (see Definition 12) is an assignment of the variables in V s.t. all constraints in Con are satisfied.

This section defined possible states of a product configuration system. How this states can occur in knowledge bases will be described in the next section.

3.3 Running Examples

This section introduces four different examples to explain how we can detect anomalies in CSPs and help knowledge engineers in their maintenance tasks.

Example 3.1 Simple consistent CSP product configuration³

The following example contains a simple notebook recommendation system denoted as CSP. The example contains variables, domains, constraints, and also products in C_P . This example will be used to explain assignment-based redundancy detection (Section 5.2.2) and the iCone interface (*intelligent environment for the development and maintenance of configuration knowledge bases*) described in Section 6.2.

$$V = \{price, cpu_cores, usage_scenario\}$$

$$D = \{ \\ \text{dom}(price) = \{399, 599, 799, 999\} \\ \text{dom}(cpu_cores) = \{2, 4\} \\ \text{dom}(usage_scenario) = \{office, multimedia, gaming\} \\ \}$$

$$C_{KB} = \{ \\ c_0 := usage_scenario = office \rightarrow (price < 599 \wedge cpu_cores = 2); \\ c_1 := usage_scenario = multimedia \rightarrow ((price < 999 \wedge cpu_cores = 4) \vee \\ \text{price} < 799); \\ c_2 := usage_scenario = gaming \rightarrow cpu_cores = 4; \\ \}$$

³This example is based on Reinfrank et al. [2015c].

$$C_P = \left\{ \begin{array}{l} (price = 399 \wedge cpu_cores = 2) \vee \\ (price = 599 \wedge cpu_cores = 4) \vee \\ (price = 799 \wedge cpu_cores = 2) \vee \\ (price = 999 \wedge cpu_cores = 4); \end{array} \right. \begin{array}{l} (p_0) \\ (p_1) \\ (p_2) \\ (p_3) \end{array}$$

$$\begin{aligned} C_R &= \emptyset \\ C &= C_{KB} \cup C_R \cup C_P \\ KB &= V \cup D \cup C \end{aligned}$$

Example 3.2 Consistent CSP product configuration⁴

The following example is denoted as CSP and shows a bike configuration knowledge base. It contains product variables as well as customer requirements. Since this example is a configuration, the set C_P is empty, s.t. each consistent combination of variable assignments is also a valid configuration. We use this example to explain redundancies (see Sections 4.1.2 and 5.2.2) and well-formedness violations (see Sections 4.1.3 and 5.2.3).

$$V = \{BikeType, FrameSize, eBike, TireWidth, UniCycle, Usage\}$$

$$D = \left\{ \begin{array}{l} dom(BikeType) = \{MountainBike, CityBike, RacerBike\}, \\ dom(FrameSize) = \{40cm, 50cm, 60cm\}, \\ dom(eBike) = \{true, false\}, \\ dom(TireWidth) = \{23mm, 37mm, 57mm\}, \\ dom(UniCycle) = \{true, false\}, \\ dom(Usage) = \{Competition, EverydayLife, HillClimbing\} \end{array} \right\}$$

⁴This example is based on Reinfrank et al. [2015a].

3 Constraint-based Product Configuration Systems

$$\begin{aligned} C_{KB} = \{ & \\ c_0 := & \text{BikeType} = \text{MountainBike} \rightarrow \text{TireWidth} > 37\text{mm} \wedge \\ & \text{FrameSize} \geq 50\text{cm}; \\ c_1 := & \text{BikeType} = \text{RacerBike} \rightarrow \text{TireWidth} = 23\text{mm} \wedge \text{FrameSize} = 60\text{cm}; \\ c_2 := & \text{BikeType} = \text{CityBike} \rightarrow \text{TireWidth} = 37\text{mm} \wedge \text{FrameSize} \geq 50\text{cm}; \\ c_3 := & \neg(\text{BikeType} \neq \text{CityBike} \wedge \text{eBike} = \text{true}); \\ c_4 := & \text{Usage} = \text{EverydayLife} \rightarrow \text{BikeType} = \text{CityBike}; \\ c_5 := & \text{Usage} = \text{HillClimbing} \rightarrow \text{BikeType} = \text{MountainBike}; \\ c_6 := & \text{Usage} = \text{Competition} \rightarrow \text{BikeType} = \text{RacerBike} \wedge \\ & \text{FrameSize} = 60\text{cm}; \\ c_7 := & \text{eBike} = \text{true} \rightarrow \text{TireWidth} = 37\text{mm}; \\ c_8 := & \text{UniCycle} = \text{false}; \\ & \} \end{aligned}$$

$$C_R = C_P = \emptyset$$

$$C = C_{KB} \cup C_P \cup C_R$$

Example 3.3 Inconsistent CSP product configuration⁵

This example is based on the previous example except the set C_R contains three customer requirements.

$$\begin{aligned} C_R = \{ & \\ c_9 : & \text{FrameSize} = 50\text{cm}; \\ c_{10} : & \text{Usage} = \text{Competition}; \\ c_{11} : & \text{eBike} = \text{true}; \\ & \} \end{aligned}$$

⁵This example is based on [Felfernig et al. \[2013a\]](#).

Example 3.4 Consistent MAUT-based product configuration⁶

We determine item orderings by using the concepts of *Multi-Attribute Utility Theory* (MAUT; Schmitt et al. [2003]; von Winterfeldt and Edwards [1986]; Keeney and Raiffa [1976]). The basic elements of MAUT are *interest dimensions* such as *profit* or *availability* describing interest focuses of a customer. For instance, *profit* denotes the performance of financial services in terms of, for example, high return rates. Furthermore, *availability* is related to aspects of accessibility of the invested sum within the targeted investment period. The following tables present an example for a MAUT-based product recommendation for financial services.

The degree to which a customer is interested in such dimensions can be derived from the articulated requirements. For example, Table 3.1 shows that our example consists of two customer variables: *investment period* and *goal*. Tables 3.2 and 3.3 include typical scoring rules (utility constraints $uc_i \in UC$) of reliable financial service providers. A customer interested in *long term investments* (*investment period* = *long term*) typically has a lower interest in *availability* than a customer who is interested in *short term investments* (*investment period* = *short term*). Similarly, customers interested in *speculations* (*goal* = *speculation*) have a lower interest in *availability* than those interested in *putting money aside for rainy days* (*goal* = *rainy days*). For the purposes of this example, we use the customer requirements specified in Table 3.4: customer *Robert* is interested in *medium term* investments with the goal of *putting money aside for rainy days*.

customer variable $v_c \in V_C$	$dom(v_c)$
v_{c1} : investment period	{long term, medium term, short term}
v_{c2} : goal	{rainy days, stable growth, speculation}

Table 3.1: Example customer variables $V_C = \{v_{c1}, v_{c2}\}$

⁶This example is based on Felfernig et al. [2013c].

3 Constraint-based Product Configuration Systems

investment period	profit	availability
short term	$4(uc_1)$	$9(uc_2)$
medium term	$6(uc_3)$	$5(uc_4)$
long term	$8(uc_5)$	$1(uc_6)$

Table 3.2: Scoring rules $\{uc_1, \dots, uc_6\} \subseteq UC$ for customer variable *investment period*.

goal	profit	availability
rainy days	$2(uc_7)$	$6(uc_8)$
stable growth	$6(uc_9)$	$4(uc_{10})$
speculation	$9(uc_{11})$	$2(uc_{12})$

Table 3.3: Scoring rules $\{uc_7, \dots, uc_{12}\} \subseteq UC$ for customer variable *goal*.

Interpreting the information of Tables 3.2, 3.3, and 3.4 we can figure out to which extent *Robert* has a focus on the interest dimensions *profit* and *availability*. *Robert* requires a *medium term investment* solution which contributes an importance of 6 to the interest dimension *profit* and an importance of 5 to the interest dimension *availability* (see Table 3.2). Furthermore, *Robert* is interested in putting money aside for *rainy days* which contributes an importance of 2 to the dimension *profit* and 6 to *availability* (see Table 3.3). Table 3.5 summarizes *Robert*'s preferences.

On the basis of such customer preferences we are able to evaluate which of a given set of alternative products (services) suits a customer's wishes and needs best. For the purpose of our simplified example we use the example products in Table 3.6, the derived product variable domains described in Table 3.7, and the recommendation result shown in Table 3.8.

We now define the dependencies between product attribute values and the interest dimensions *profit* and *availability*. For instance, financial services including shares support a higher profit (see Table 3.9) whereas a higher value fluctuation leads to a higher profit utility value (see Table 3.10). Furthermore, financial services without shares have a higher *availability*, and those with a higher value fluctuation have a higher (potential) *profit*.

3.3 Running Examples

customer	investment period	goal
Robert	medium term (r_1)	rainy days (r_2)

Table 3.4: Example customer requirements $C_R = \{r_1, r_2\}$.

customer	profit	availability
Robert	$6 + 2 = 8$	$5 + 6 = 11$

Table 3.5: Interests of customer *Robert* derived from the constraints in Tables 3.2, 3.3, and 3.4.

Interpreting the constraints of Tables 3.8, 3.9, and 3.10, we can derive product assortment specific scoring rules $up_i \in UC$ (Tables 3.11 and 3.12).

Finally, our example also consists of some restrictions. Those restrictions remove some of the products in C_P for a recommendation task whereas products with a low utility will be presented to a customer at the end of the list of recommended products. Table 3.13 gives an overview of the constraints in the example knowledge base and explains why the product *equity* is not part of the resulting product list.

On the basis of these scoring rules we can determine the extent to which our financial services contribute to the interest dimensions *profit* and *availability* (see Table 3.12). Exploiting the identified product utilities, we can determine the customer-specific utility of each product *prod* which is contained in the recommendation result (see Table 3.12). The utility of a product or service can be determined on the basis of Equation 3.1

$$utility(x) = \sum_{i=1}^n in_i \times con_i(prod) \quad (3.1)$$

where $utility(prod)$ specifies the overall utility of a product/service *prod* for a specific customer. The overall utility of *prod* is defined as sum over the customer's interest in dimension i (in_i) times the contribution of product

3 Constraint-based Product Configuration Systems

constraint	name	shares	value fluctuation
$prod_1$	balanced funds	50%	medium
$prod_2$	bonds	0%	medium
$prod_3$	bonds2	0%	high
$prod_4$	equity	100%	very high

Table 3.6: Example set of financial services represented by $C_P = \{prod : prod_1 \vee prod_2 \vee prod_3 \vee prod_4\}$ where $prod_1 : name = \text{balanced funds} \wedge shares = 50\% \wedge value\ fluctuation = \text{medium}$; $prod_2 : name = \text{bonds} \wedge shares = 0\% \wedge value\ fluctuation = \text{medium}$; $prod_3 : name = \text{bonds2} \wedge shares = 0\% \wedge value\ fluctuation = \text{high}$; and $prod_4 : name = \text{equity} \wedge shares = 100\% \wedge value\ fluctuation = \text{very high}$;

product variable $v_p \in V_P$	$dom(v_p)$
$v_{p1} : name$	{balanced funds, bonds, bonds2, equity}
$v_{p2} : shares$	{0%, 50%, 100%}
$v_{p3} : value\ fluctuation$	{low, medium, high, very high}

Table 3.7: Example product variables $V_P = \{v_{p1}, v_{p2}, v_{p3}\}$

$prod$ (in our case financial service) to dimension i (con_i). In our example, *balanced funds* have a higher utility for *Robert* than *bonds* and *bonds2* (see Table 3.15).

In order to test whether a given set of utility constraints calculates intended rankings, a corresponding set of *examples* (test cases) can be provided by marketing and sales experts (see, Table 3.16). In the case that the rankings calculated by the utility constraint set are in contradiction with the rankings of the given examples, we have to identify repairs such that the consistency with the examples is restored. In our scenario, the examples $E = \{e_1, e_2, e_3\}$ are partially contradicting the rankings (utilities) shown in Table 3.15 (e.g., the utility of *bonds* is lower than the utility of *balanced funds* if a customer is interested in medium term investments for rainy days, the contrary is specified in $e_1 : utility(bonds) > utility(balanced\ funds)$). Consequently, we have to identify an adaptation of our utility constraints. An approach to derive such adaptations automatically will be discussed in the following.

3.3 Running Examples

variable in V	$prod_1$	$prod_2$	$prod_3$
v_{c1} : investment period	medium term	medium term	medium term
v_{c2} : goal	rainy days	rainy days	rainy days
v_{p1} : name	balanced funds	bonds	bonds2
v_{p2} : shares	50%	0%	100%
v_{p3} : value fluctuation	medium	medium	high

Table 3.8: Recommendation result for the constraint-based product recommendation problem.

shares	profit	availability
0%	2	7
50%	5	5

Table 3.9: Scoring rules for product variable *shares*.

value fluctuation	profit	availability
medium	5	6
high	7	4

Table 3.10: Scoring rules for product variable *value fluctuation*.

name	profit	availability
balanced funds	$5(up_1)$	$5(up_2)$
bonds	$2(up_3)$	$7(up_4)$
bonds2	$2(up_5)$	$7(up_6)$

Table 3.11: Product-specific scoring rules $\{up_1, \dots, up_6\} \subseteq UC$ for product attribute *shares*.

name	profit	availability
balanced funds	$5(up_7)$	$6(up_8)$
bonds	$5(up_9)$	$6(up_{10})$
bonds2	$7(up_{11})$	$4(up_{12})$

Table 3.12: Product-specific scoring rules $\{up_7, \dots, up_{12}\} \subseteq UC$ for product attribute *value fluctuation*.

3 Constraint-based Product Configuration Systems

filter constraint $c \in C_{KB}$
$c_1 : \text{goal} = \text{rainy days} \rightarrow \text{shares} \neq 100\%$;
$c_2 : \text{investment period} = \text{shortterm} \rightarrow \text{value fluctuation} \neq \text{high}$;
$c_3 : \neg(\text{goal} = \text{stable growth} \wedge \text{investment period} = \text{short term})$;
$c_4 : \neg(\text{goal} = \text{rainy days} \wedge \text{investment period} = \text{long term})$;

Table 3.13: Example constraints $C_{KB} = \{c_1, c_2, c_3, c_4\}$.

name	profit	availability
balanced funds	$5 + 5 = 10$	$5 + 6 = 11$
bonds	$2 + 5 = 7$	$7 + 6 = 13$
bonds2	$2 + 7 = 9$	$7 + 4 = 11$

Table 3.14: Utilities of products regarding interest dimensions (result of interpreting Tables 3.11 and 3.12).

customer	product	profit	availability	utility
Robert	balanced funds	8×10	11×11	201
	bonds	8×7	11×13	199
	bonds2	8×9	11×11	193

Table 3.15: Utilities of products for customer *Robert*.

example	investment period	goal	ranking
e_1	medium term	for rainy days	$utility(bonds) > utility(balanced\ funds)$
e_2	medium term	for rainy days	$utility(bonds2) > utility(balanced\ funds)$
e_3	medium term	for rainy days	$utility(bonds) > utility(bonds2)$

Table 3.16: Examples $E = \{e_1, e_2, e_3\}$ of intended service orderings.

Now we have defined consistent (Examples 3.1 and 3.2) and inconsistent (Example 3.3) constraint-based product configuration problems and a MAUT-based product configuration (Example 3.4). We will use the last example in Section 4.3 for the automated repair of scoring rules. The other

3.3 Running Examples

examples will be used in the next chapter to detect (Section 4.1), explain (Section 4.2), and verify the understandability (Section 4.4) of anomalies.

4 Anomaly Management for Constraint-based Product Configuration Systems

In this chapter we give a detailed description of the term *anomaly management*. Therefore, we first describe the different types of anomalies (Section 4.1). While detecting algorithms is a well-researched area, we extend the term management by explaining anomalies (Section 4.2) and generate repair actions for MAUT-based product configurations (section 4.3). Finally, we give hints for presenting conflicts and diagnoses to users (Section 4.4).

4.1 Types of Anomalies in Constraint-based Product Configuration Systems

In this section we describe the principles of anomalies. We explain conflicts and diagnoses (Section 4.1.1), redundancies (Section 4.1.2), and well-formedness violations (Section 4.1.3).

4.1.1 Conflicts and Diagnoses¹

In this section we give a detailed overview of principles of conflicts and diagnoses. Based on the definitions in Section 3.2 we first give a detailed

¹This Section is based on [Felfernig et al. \[2014e\]](#).

4 Anomaly Management for Constraint-based Product Configuration Systems

overview of conflicts and diagnoses and the calculation of all conflicts and diagnoses based on HSDAG.

As explained in the previous Sections we are looking for consistent configuration knowledge-bases (see Definition 3). If a knowledge base is inconsistent, we can look for minimal conflicts (see Definition 5) and / or minimal diagnoses (see Definition 7). To get useable conflicts and diagnoses we have to differ between a set of constraints which can be diagnosed (e.g., customer requirements) and a set of constraints that shouldn't be part of diagnoses (e.g., the set of constraints which can not be changed during a customer session). Therefore, we introduced the set C_{KB} for not diagnosable constraints and the set C_R for constraints that can be part of conflicts and diagnoses.

When we focus on the set C_R and say that C_{KB} is consistent, our example 3.3 contains two minimal conflict sets: $CS_1 = \{c_{10}\}$ because it is not possible to have an *ebike* for *competition* with a frame size of 50cm and $CS_2 = \{c_9, c_{11}\}$ as bikes used for *competition* do not support *eBikes* with a frame size of 50cm.

Note that we are only focusing on minimal sets of conflicts and diagnoses because we assume that repair actions with minimal constraint sets a) have the lowest impact on the behavior of the knowledge base and b) such sets are mostly easier to understand and maintain for knowledge engineers and customers.

To get many / all sets which have the property of being a diagnosis / conflict we use the HSDAG algorithm [Reiter, 1987]. The algorithm uses the output of a conflict / diagnosis detection algorithm and splits the output (a set of constraints) into its constraints. Each of the constraints will be shifted from C_R to C_{KB} and then the conflict / diagnosis detection algorithm runs again. If the algorithm returns an empty set, we can say that there are no further minimal conflict / diagnosis sets. The branches in the HSDAG

4.1 Types of Anomalies in Constraint-based Product Configuration Systems

algorithm can be expanded e.g., with breadth-first search, depth-first search or iterative deepening. An example - based on our example in Section 3.3 - is given in Figure 4.1. A detailed discussion about HSDAG is given in Felfernig and Schubert [2011b]; Reiter [1987].

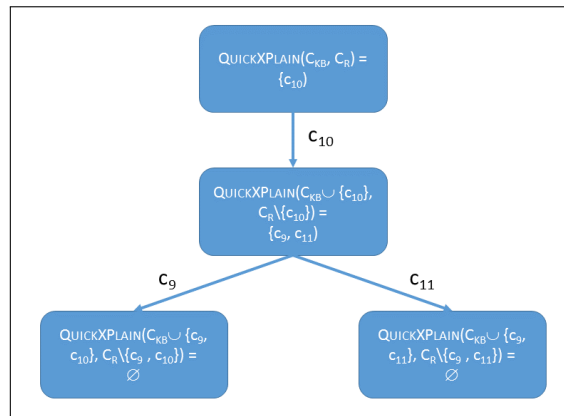


Figure 4.1: Graphical representation of all conflicts in Example 3.3 calculated with HSDAG [Reiter, 1987] and QuickXPlain [Junker, 2004].

If we can guarantee that a knowledge base is conflict-free, we can also detect redundancies and well-formedness violations. The principles of these violations will be explained in the next sections.

4.1.2 Redundancies²

In this Section we focus on situations where knowledge engineers are defining redundant constraints that, when deleted from the constraint set, do not change the semantics of the remaining constraints. This might happen for performance reasons or due to the fact that knowledge engineers are not aware of the redundancy. We focus on the latter situation.

²This Section is based on Felfernig et al. [2014d].

4 Anomaly Management for Constraint-based Product Configuration Systems

Formally, a redundancy can be described as follows: if $C = \{c_1, c_2, \dots, c_n\}$ is a set of constraints and one constraint $c_i \in C$ is redundant, then $C \setminus \{c_i\} \cup \bar{C}$ is inconsistent. In this context, \bar{C} is the negation³ of C : if $C = \{c_1, c_2, \dots, c_n\}$ then $\bar{C} = \{\neg c_1 \vee \neg c_2 \vee \dots \vee \neg c_n\}$.

Redundancy elimination in knowledge bases is a topic extensively investigated by AI research. Redundant constraints play a role, for example, in the development and maintenance of configuration knowledge bases [Sabin and Freuder, 1998]. The authors introduce concepts for the detection of redundant constraints in conditional constraint satisfaction problems (CCSPs). The approach is based on the idea of analyzing the solution space of the given problem (on the level of individual solutions) in order to detect different types of redundant constraints. Piette [2008] provides a discussion of the role of redundancy elimination in SAT solving. The author introduces an (incomplete) algorithm for the elimination of redundant clauses and shows its applicability on the basis of an empirical study. The role of redundancies in ontology development is analyzed by Fahad and Qadir [2008]. The authors point out the importance of redundancy elimination and discuss typical modeling errors that occur during ontology development and maintenance. Grimm and Wissmann [2011] introduce algorithms for redundancy elimination in OWL ontologies. The authors propose an algorithm that computes redundant axioms by exploiting prior knowledge of the concepts of *minimal derivation trees* which do not include any pair of identical atoms where one is the predecessor of the other one.

All the mentioned approaches focus on the identification of redundant constraints in centralized scenarios where one or a few knowledge engineers are interested in identifying redundant constraints. In such scenarios it is assumed that only a small subset of the given constraints is redundant (this assumption is also denoted as *low redundancy assumption*; Grimm and Wissmann [2011]). We go one step further and propose an algorithm that is especially useful in distributed knowledge engineering scenarios where we can expect a *larger number of redundant constraints*, due to the fact that

³We denote the negation of $C(\bar{C})$ also the *complement* of C .

4.1 Types of Anomalies in Constraint-based Product Configuration Systems

different contributors add constraints that are related to the same topic (see Section 5.2, Chklovski and Gil [2005]; Richardson and Domingos [2003]). We denote the assumption of larger sets of redundant constraints the *high redundancy assumption*. For example, we envision a scenario where a large number of users propose constraints to be applied by a constraint-based configuration recommendation engine [Felfernig and Burke, 2008] and the task of an underlying algorithm is to identify minimal sets of constraints that retain the semantics of the original constraint set. We denote such constraint sets as *minimal cores*. Note that the following discussions are based on the assumption of *consistent constraint sets* (see Definition 3). Methods for consistency restoration are discussed in section 5.2.1.

In this thesis we extend current state-of-the-art in detecting redundant constraints by detecting sets of redundant constraints. We call this algorithm *CoreDiag* and explain the algorithm in section 5.2.2.

4.1.3 Well-formedness Violations⁴

Well-formedness violations focus on the variables and domains of a knowledge base instead of constraints. Such anomalies are well-discussed in other research disciplines like feature models but are not well researched in the area of constraint-based product configuration. In the following, we give an overview of violations in product configurations.

- **dead domain elements:** a domain element can never be part of a consistent configuration (see Definition 9). Such domain elements can be removed from the domain before doing any calculations.
- **full mandatory domain element:** contrary to dead domain elements, a full mandatory domain element has to be in a consistent product configuration (see Definition 10). If a domain element is full mandatory, we can say that all other domain elements of this variable are dead. Such variables can be removed before doing consistency checks.

⁴This Section is based on Felfernig et al. [2013a] and Benavides et al. [2013, 2010].

4 Anomaly Management for Constraint-based Product Configuration Systems

Afterwards the variable and the consistent domain element can be added again as a constant.

- unnecessary refinement: The third type of well-formedness violation compares the occurrences of domain elements between two variables (see Definition 11). For example, if only *RacerBikes* can have carbon material and *CityBikes* and *MountainBikes* can only have aluminium frames (and vice versa), we can say that the *framematerial* is an unnecessary refinement of the *Usage* variable. We can say that we can ignore the variable *framematerial* and the corresponding domain and constraints in our algorithms, since that variable does not change the behavior of the configuration knowledge base.

While some of the detected anomalies in a knowledge base are easy to understand, others may be complex and difficult to understand. In the next section we show, how we can explain anomalies to reduce the repair time.

4.2 Anomaly Explanation⁵

How to explain the anomalies from Section 4.1 will be shown in this Section. An overview of these anomalies and related property checks is shown in Table 4.1. Note that we are using a product configuration example in this section but we can also extend the presented explanations to knowledge-based recommendation systems by adding the set C_P to the set of not diagnosable constraints.

Conflicts and Diagnoses. If constraints in the knowledge base C_{KB} are inconsistent ($inconsistent(C_{KB})$), we are interested in solutions to the product configuration diagnosis. In this case we want to figure out which are the minimal sets of constraints that are responsible for the given inconsistency in the product configuration. The product configuration of our example 3.3

⁵This Section is based on [Felfernig et al. \[2013a\]](#).

4.2 Anomaly Explanation

is an example of a consistent product configuration. For conflicts we can say that a diagnosis can explain a conflict. Our second example in Section 3.3 contains two conflict sets $CS_1 = \{c_9, c_{10}\}$ and $CS_2 = \{c_{10}, c_{11}\}$. We can explain CS_1 by calculating at least one diagnosis for our knowledge base C_{KB} . $\Delta_1 = \{c_1, c_6\}$ explains CS_1 since those constraints forbid the combination of *RacerBikes* with a *FrameSize* of 50cm.

Redundant constraint. In our working example the constraint c_7 is redundant since only *CityBikes* can be *eBikes* (c_3) and have *tires* with a width of 37mm (c_2). If we check the consistency of $C_{KB} \setminus \{c_7\} \cup \neg C_{KB}$, we see that c_7 is redundant since the expression is inconsistent. In other words, $C_{KB} \setminus \{c_7\} \models c_7$, i.e., c_7 logically follows from $C_{KB} \setminus \{c_7\}$ – therefore it is redundant. Consequently, the constraint set $\{c_7\}$ can be deleted from the product configuration knowledge base without changing the underlying semantics. To check why the constraint c_7 is redundant, we have to compare the set of constraints without the diagnosable constraint $C_{KB} \setminus \{c_7\}$ with the negation of the knowledge base $\overline{C_{KB}}$, s.t., $C_{KB} \setminus \{c_7\} \cup \overline{C_{KB}} = \{c_2, c_3\}$.

Dead domain element. If a domain element $d_i \in \text{dom}(v_j)$ is not included in any of the possible configurations (i.e., $\text{inconsistent}(C_{KB}, C_R \cup \{v_j = d_i\})$; see Definition 9), we are interested in solutions to the product configuration diagnosis task $(C_{KB}, C_R \cup \{v_j = d_i\})$. We are able to figure out the minimal sets of constraints that are responsible for the non-acceptance of $v_j = d_i$. The assignments *FrameSize* = 40cm and *UniCycle* = true; can never be part of a consistent instance because *MountainBikes* and *CityBikes* require at least 50cm and *RacerBikes* require a *FrameSize* of 60cm and our current knowledge base does not support *UniCycles*. If we then want to make *UniCycle* = true a domain element which is included in at least one configuration, the diagnosis for $(C_{KB}, C_R \cup \{\text{UniCycle} = \text{true}\})$ is $\Delta_1 = \{c_8\}$.

Full mandatory domain element. A domain element $d_i \in \text{dom}(v_j)$ is fully mandatory if it is included in every possible solution (void configuration; see Definition 10), i.e., $\text{inconsistent}(C_{KB}, C_R \cup v_j = d_i)$. If we want to adapt the product configuration in such a way that it also allows $v_j = d_i$ to be not

4 Anomaly Management for Constraint-based Product Configuration Systems

included, we can determine the corresponding (minimal) sets of responsible constraints by solving the product configuration diagnosis task $(C_{KB}, C_R \cup \{v_j = d_i\})$. The knowledge base can never be consistent if $UniCycle \neq false$. In that case we can say that the domain element $false$ of the domain $dom(UniCycle)$ is full mandatory and $UniCycle = true$ can never be in a consistent knowledge base (dead domain element). If we want to allow configurations where $UniCycle \neq false$ is included, the only diagnosis for $(C_{KB}, C_R \cup \{UniCycle = true\})$ is $\Delta 1 = c_8$.

Unnecessary refinement. Such an unnecessary refinement consists of two variables. If the first variable has an assignment, it is possible to predict the assignment of the second variable because the second variable can only have exactly one consistent assignment. A formal definition is given in Definition 13. In our example the variable pair $Usage$ and $BikeType$ is unnecessarily refined because whenever $Usage = EverydayLife$ the $BikeType$ has to be $CityBike$, $Usage = HillClimbing$ always leads to $BikeType = MountainBike$, and $Usage = Competition$ is always combined with the assignment $BikeType = RacerBike$. If such a violation occurs, we can recommend the knowledge engineer to remove the variable $Usage$ and replace it with the variable $BikeType$ in the constraints. Adding a constraint $c_i := \neg(Usage = EverydayLife \rightarrow BikeType = CityBike) \wedge \neg(Usage = HillClimbing \rightarrow BikeType = MountainBike) \wedge \neg(Usage = Competition \rightarrow BikeType = RacerBike)$ to C_{KB} and running $QuickXPlain(C_R, C_{KB})$, returns a set of constraints which explains why other assignments for the variables lead to inconsistencies. In our example a diagnosis for $C_R \cup C_{KB}$ would be $\{c_4, c_5, c_6\}$.

The two basic algorithms for determining diagnoses and redundancies are FASTDIAG and FMCORE. FASTDIAG [Felfernig et al., 2012b] is a divide-and-conquer algorithm that supports the efficient determination of minimal diagnoses without the need of having conflict sets available. FMCORE is an algorithm which focuses on the determination of minimal cores, i.e., redundancy-free subsets of a constraint set.

4.3 Automated Repair of Scoring Rules

In FASTDIAG (see Algorithm 3) the set C_R represents the set of constraints where a diagnosis should be searched. The set C_{KB} contains all constraints which can not be part of a diagnosis. For example, if we want to diagnose a configuration ($C_{KB} \cup C_R$ is inconsistent – see Table 4.1), we will activate the algorithm with $FASTDIAG(C_{KB}, C_R)$.

Now we can explain anomalies to knowledge engineers and users of constraint-based configuration and knowledge-based recommendation systems.

In this Section we described how we can calculate explanations for anomalies to knowledge engineers and users of constraint-based product configuration systems. The following section describes how repair actions can be done automatically in MAUT based configuration systems.

4.3 Automated Repair of Scoring Rules⁶

Products included in a recommendation have to be ranked according to their relevance for the customer [Felfernig et al., 2006a; Felfernig and Burke, 2008]. In the line of serial position effects which induce customers to preferably take a look at and select items at the beginning of a list, the high-ranking of the most relevant items is extremely important [Lashley, 1951; Gershberg and Shimamura, 1994]. For the determination of such rankings we apply the concepts of Multi-Attribute Utility Theory (MAUT) [Winterfeldt and Edwards, 1986; Keeney and Raiffa, 1993; Schmitt et al., 2003] where each product is evaluated according to a predefined set of interest dimensions which are abstract evaluation criteria for products. Profit and availability are examples for such interest dimensions in the domain of financial services. For example, if a customer is interested in high return rates and long term investments, the dimension profit is very

⁶This Section is based on Felfernig et al. [2013c].

Analysis operation	Property Check	Explanation (Diagnosis Task)
Void feature model	$inconsistent(C_{KB})?$	$FASTDIAG(\emptyset, C_{KB})$
Dead domain element ($d_i \in dom(v_j)$)	$inconsistent(C_{KB} \cup \{v_j = d_i\})?$	$FASTDIAG(\{v_j = d_i\}, C_{KB})$
Full mandatory ($d_i \in dom(v_j)$)	$inconsistent(C_{KB} \cup \{\neg v_j = d_i\})?$	$FASTDIAG(\{\neg v_j = d_i\}, C_{KB})$
Unnecessary refinement (v_i, v_j)	$inconsistent(C_{KB} \cup \{ \neg(v_i = val_a \wedge v_j = val_a) \vee \neg(v_i = val_b \wedge v_j = val_e) \vee \dots \vee \neg(v_i = val_c \wedge v_j = val_f) \})?$	$FASTDIAG(\{ \neg(v_i = a \wedge v_j = d) \wedge \neg(v_i = b \wedge v_j = e) \wedge \dots \wedge \neg(v_i = c \wedge v_j = f) \}, C_{KB})$
Redundant (c_i)	$inconsistent((C_{KB} \setminus \{c_i\}) \cup \overline{C_{KB}})?$	$c_i \notin FMCore(C_{KB})$

Table 4.1: Product configuration analysis operations, property checks, and related explanations. For example, figuring out whether a configuration is inconsistent (no solution can be found) can be determined on the basis of a consistency check ($inconsistent(C)$). A related explanation can be determined by solving the diagnosis task (C_{KB}, C_R). The related diagnosis ($FASTDIAG$) and redundancy detection algorithm ($FMCore$) are discussed in Section 5.2.

4.3 Automated Repair of Scoring Rules

important. Consequently, customer requirements influence the importance of corresponding interest dimensions.

The consistency between utility constraints (scoring rules) and a company's marketing and sales strategy plays an important role for the successful application of recommender technologies. These constraints have to reflect marketing and sales strategies (in our example 3.4 those of financial service providers). Experiences from commercial projects [Felfernig et al., 2006a] show a remarkable need for a knowledge acquisition support that alleviates the development and maintenance of utility constraint sets. The manual adaptation of utility constraints is a time-consuming and error-prone task since such constraints are strongly interdependent. Therefore, we developed techniques which support knowledge engineers in the identification and repair of faulty elements in utility constraint sets. In this section we transform the table representation of the MAUT recommendation problem into a CSP, present adaptation concepts which automatically identify the sources of inconsistencies in utility constraint sets, and propose corresponding repair actions. The presented approach has been implemented for a commercial recommender environment [Felfernig et al., 2006a] and is in the line of previous work [Felfernig and Shchekotykhin, 2006] related to effective knowledge acquisition interfaces for recommender applications.

Utility Constraint Set

We now transform our utility constraint set (tabular representation; see Example 3.4) into a corresponding constraint-based representation which is used as input for solving a non-linear optimization problem (see Fourer et al. [2002]). Regarding the definitions of Tables 3.2 and 3.3, we introduce the following set of utility constraints related to the required investment period and the customer's personal goals. For instance, constraint uc_1 denotes the fact that for customers requiring financial services with short term investment periods, the dimension profit is of medium importance on a value scale of [1..10], whereas availability aspects play a significantly more

4 Anomaly Management for Constraint-based Product Configuration Systems

important role (uc_2). $\{uc_1, \dots, uc_6\}$ represent the utility definitions of Table 3.2, $\{uc_7, \dots, uc_{12}\}$ represent the definitions of Table 3.3.

$$\begin{aligned}
 uc_1 &: profit(investmentperiod_{short}) = 4 \\
 uc_2 &: availability(investmentperiod_{short}) = 9 \\
 uc_3 &: profit(investmentperiod_{medium}) = 6 \\
 uc_4 &: availability(investmentperiod_{medium}) = 5 \\
 uc_5 &: profit(investmentperiod_{long}) = 8 \\
 uc_6 &: availability(investmentperiod_{long}) = 1 \\
 uc_7 &: profit(goal_{rainydays}) = 2 \\
 uc_8 &: availability(goal_{rainydays}) = 6 \\
 uc_9 &: profit(goal_{growth}) = 6 \\
 uc_{10} &: availability(goal_{growth}) = 4 \\
 uc_{11} &: profit(goal_{speculation}) = 9 \\
 uc_{12} &: availability(goal_{speculation}) = 2
 \end{aligned}$$

We denote each constraint defining such utility values as utility constraint $uc_i \in UC$. Since we are interested in a utility constraint set which is consistent with all the examples $e_i \in E$, we have to check the consistency of the given set of utility constraints with $\bigcup e_i$. This type of consistency check requires a representation where each example is described by a separate set of finite domain variables. For instance, the contribution to profit provided by the customer attribute investment period in example e_1 is stored in the variable $profit(investmentperiod_{e_1})$. The following representation of examples can be directly interpreted by a non-linear optimization algorithm [Fourer et al., 2002].

$$\begin{aligned}
 e_1 &: profit(investmentperiod_{e_1}) = profit(investmentperiod_{medium}) \wedge \\
 &availability(investmentperiod_{e_1}) = availability(investmentperiod_{medium}) \wedge \\
 &profit(goal_{e_1}) = profit(goal_{rainydays}) \wedge \\
 &availability(goal_{e_1}) = availability(goal_{rainydays}) \wedge \\
 &utility(balancedfunds_{e_1}) < utility(bonds_{e_1})
 \end{aligned}$$

$$\begin{aligned}
 e_2 &: profit(investmentperiod_{e_2}) = profit(investmentperiod_{medium}) \wedge \\
 &availability(investmentperiod_{e_2}) = availability(investmentperiod_{medium}) \wedge
 \end{aligned}$$

4.3 Automated Repair of Scoring Rules

$$\begin{aligned} \text{profit}(\text{goal}_{e_2}) &= \text{profit}(\text{goal}_{\text{rainydays}}) \wedge \\ \text{availability}(\text{goal}_{e_2}) &= \text{availability}(\text{goal}_{\text{rainydays}}) \wedge \\ \text{utility}(\text{balancedfunds}_{e_2}) &< \text{utility}(\text{bonds2}_{e_2}) \end{aligned}$$

$$\begin{aligned} e_3 : \text{profit}(\text{investmentperiod}_{e_3}) &= \text{profit}(\text{investmentperiod}_{\text{medium}}) \wedge \\ \text{availability}(\text{investmentperiod}_{e_3}) &= \text{availability}(\text{investmentperiod}_{\text{medium}}) \wedge \\ \text{profit}(\text{goal}_{e_3}) &= \text{profit}(\text{goal}_{\text{rainydays}}) \wedge \\ \text{availability}(\text{goal}_{e_3}) &= \text{availability}(\text{goal}_{\text{rainydays}}) \wedge \\ \text{utility}(\text{bonds2}_{e_3}) &< \text{utility}(\text{bonds}_{e_3}) \end{aligned}$$

The overall customer interest in the dimension profit is stored in $\text{profit}(e_i)$. The values of these variables represent the sum over all defined contributions of customer requirements of example e_i to the dimension profit. This approach is analogously applied to the dimension availability ($\text{availability}(e_i)$). We denote constraints summing up customer utilities as $s_i \in S$. The following constraints implement the definitions for e_1, e_2 , and e_3 .

$$s_1 : \text{profit}(e_1) = \text{profit}(\text{investmentperiod}_{e_1}) + \text{profit}(\text{goal}_{e_1})$$

$$s_2 : \text{availability}(e_1) = \text{availability}(\text{investmentperiod}_{e_1}) + \text{availability}(\text{goal}_{e_1})$$

$$s_3 : \text{profit}(e_2) = \text{profit}(\text{investmentperiod}_{e_2}) + \text{profit}(\text{goal}_{e_2})$$

$$s_4 : \text{availability}(e_2) = \text{availability}(\text{investmentperiod}_{e_2}) + \text{availability}(\text{goal}_{e_2})$$

$$s_5 : \text{profit}(e_3) = \text{profit}(\text{investmentperiod}_{e_3}) + \text{profit}(\text{goal}_{e_3})$$

$$s_6 : \text{availability}(e_3) = \text{availability}(\text{investmentperiod}_{e_3}) + \text{availability}(\text{goal}_{e_3})$$

4 Anomaly Management for Constraint-based Product Configuration Systems

For each service part of our example assortment we specify its contribution to the given interest dimensions. For instance, the shares percentage specified for the service balanced funds defines an average interest in the dimension profit. In our CSP, we define this fact as

$$profitshares(balancedfunds) = 5.$$

Analogously, we define the relationship between the interest dimension availability and shares percentage as

$$availabilityshares(balancedfunds) = 5.$$

We denote each constraint defining a utility value for a certain product (service) as utility constraint $up_i \in UC$. The following constraints implement the definitions of Tables 3.11 and 3.12.

$$\begin{aligned} up_1 &: profitshares(balancedfunds) = 5 \\ up_2 &: availabilityshares(balancedfunds) = 5 \\ up_3 &: profitshares(bonds) = 2 \\ up_4 &: availabilityshares(bonds) = 7 \\ up_5 &: profitshares(bonds2) = 2 \\ up_6 &: availabilityshares(bonds2) = 7 \\ up_7 &: profitfluctuation(balancedfunds) = 5 \\ up_8 &: availabilityfluctuation(balancedfunds) = 6 \\ up_9 &: profitfluctuation(bonds) = 5 \\ up_{10} &: availabilityfluctuation(bonds) = 6 \\ up_{11} &: profitfluctuation(bonds2) = 7 \\ up_{12} &: availabilityfluctuation(bonds2) = 4 \end{aligned}$$

For each $uc_i \in UC$ (and each $up_i \in UC$) we add a corresponding repair constraint $cr_i (pr_i)$ which specifies possible repairs for $uc_i (up_i)$. The idea behind repair constraints is that if the utility constraint set is inconsistent with the examples, a non-linear optimization process can identify minimal repairs for $uc_i (up_i)$ which are within the boundaries defined by repair

4.3 Automated Repair of Scoring Rules

constraints. These repairs should change the original uc_i (up_i) as little as possible.⁷ Therefore, we define an interval for the accepted changes for each $uc_i \in C$ and each $up_i \in P$. Each of the following example repair constraints allows changes of the given evaluations by at most one unit. We denote $\bigcup cr_i \cup \bigcup pr_i$ as set of repair constraints R .

- $cr_1 : profit(investmentperiod_{short}) \in \{3,4,5\}$
- $cr_2 : availability(investmentperiod_{short}) \in \{8,9,10\}$
- $cr_3 : profit(investmentperiod_{medium}) \in \{5,6,7\}$
- $cr_4 : availability(investmentperiod_{medium}) \in \{4,5,6\}$
- $cr_5 : profit(investmentperiod_{long}) \in \{7,8,9\}$
- $cr_6 : availability(investmentperiod_{long}) \in \{0,1,2\}$
- $cr_7 : profit(goal_{rainydays}) \in \{1,2,3\}$
- $cr_8 : availability(goal_{rainydays}) \in \{5,6,7\}$
- $cr_9 : profit(goal_{growth}) \in \{5,6,7\}$
- $cr_{10} : availability(goal_{growth}) \in \{4,5,6\}$
- $cr_{11} : profit(goal_{speculation}) \in \{8,9,10\}$
- $cr_{12} : availability(goal_{speculation}) \in \{1,2,3\}$
- $pr_1 : profitshares(balancedfunds) \in \{4,5,6\}$
- $pr_2 : availabilityshares(balancedfunds) \in \{4,5,6\}$
- $pr_3 : profitshares(bonds) \in \{1,2,3\}$
- $pr_4 : availabilityshares(bonds) \in \{6,7,8\}$
- $pr_5 : profitshares(bonds2) \in \{1,2,3\}$
- $pr_6 : availabilityshares(bonds2) \in \{6,7,8\}$
- $pr_7 : profitfluctuation(balancedfunds) \in \{4,5,6\}$
- $pr_8 : availabilityfluctuation(balancedfunds) \in \{5,6,7\}$
- $pr_9 : profitfluctuation(bonds) \in \{4,5,6\}$
- $pr_{10} : availabilityfluctuation(bonds) \in \{5,6,7\}$
- $pr_{11} : profitfluctuation(bonds2) \in \{6,7,8\}$
- $pr_{12} : availabilityfluctuation(bonds2) \in \{3,4,5\}$

The profit of a financial service is defined by the sum of contributions of the values of shares and value fluctuation. Availability of a service is as well

⁷Note that changes of at most one unit are only introduced for this example, the range of possible changes is more flexible. In the current implementation it can be specified by knowledge engineers.

4 Anomaly Management for Constraint-based Product Configuration Systems

defined by the sum of related contributions. We denote each rule summing up service utility values as $s_i \in S$. The following constraints implement the definitions of Table 3.14.

$$\begin{aligned} s7 : profit(balancedfunds) &= \\ & profitshares(balancedfunds) + profitfluctuation(balancedfunds) \\ s8 : profit(bonds) &= \\ & profitshares(bonds) + profitfluctuation(bonds) \\ s9 : profit(bonds2) &= \\ & profitshares(bonds2) + profitfluctuation(bonds2) \\ s10 : availability(balancedfunds) &= \\ & availabilityshares(balancedfunds) + \\ & availabilityfluctuation(balancedfunds) \\ s11 : availability(bonds) &= \\ & availabilityshares(bonds) + availabilityfluctuation(bonds) \\ s12 : availability(bonds2) &= \\ & availabilityshares(bonds2) + availabilityfluctuation(bonds2) \end{aligned}$$

The following constraints specify the calculation of product utilities, where $utility(prod_{e_i})$ specifies the utility of product $prod$ in the context of example e_i .

$$\begin{aligned} s13 : utility(balancedfunds_{e_1}) &= profit(balancedfunds) \times profit(e_1) + \\ & availability(balancedfunds) \times availability(e_1) \\ s14 : utility(bonds_{e_1}) &= profit(bonds) \times profit(e_1) + \\ & availability(bonds) \times availability(e_1) \\ s15 : utility(bonds2_{e_1}) &= profit(bonds2) \times profit(e_1) + \\ & availability(bonds2) \times availability(e_1) \\ s16 : utility(balancedfunds_{e_2}) &= profit(balancedfunds) \times profit(e_2) + \\ & availability(balancedfunds) \times availability(e_2) \\ s17 : utility(bonds_{e_2}) &= profit(bonds) \times profit(e_2) + \\ & availability(bonds) \times availability(e_2) \\ s18 : utility(bonds2_{e_2}) &= profit(bonds2) \times profit(e_2) + \\ & availability(bonds2) \times availability(e_2) \\ s19 : utility(balancedfunds_{e_3}) &= profit(balancedfunds) \times profit(e_3) + \\ & availability(balancedfunds) \times availability(e_3) \end{aligned}$$

4.3 Automated Repair of Scoring Rules

$$\begin{aligned} s_{20} : & \text{utility}(\text{bonds}_{e_3}) = \text{profit}(\text{bonds}) \times \text{profit}(e_3) + \\ & \text{availability}(\text{bonds}) \times \text{availability}(e_3) \\ s_{21} : & \text{utility}(\text{bonds2}_{e_3}) = \text{profit}(\text{bonds2}) \times \text{profit}(e_3) + \\ & \text{availability}(\text{bonds2}) \times \text{availability}(e_3) \end{aligned}$$

Automated Repair of Scoring Rules

All the mentioned constraints are constituting elements of a corresponding nonlinear optimization problem [Fourer et al., 2002] which represents a Constraint Set Adaptation Problem (see Definition 12).

The constraints *Con* defined in the previous part of this section are the basic elements of an optimization problem of minimizing repair distances between original scoring values and corresponding repair values using the following minimization function - see Equation 4.1.

$$\text{Minimize} : \sum_{i=1}^m |\text{val}(uc_i) - \text{val}(cr_i)| + \sum_{j=1}^n |\text{val}(up_j) - \text{val}(pr_j)| \quad (4.1)$$

In this formula $|\text{val}(uc_i) - \text{val}(cr_i)|$ denotes the degree to which the original value of the utility constraint (scoring value for customer requirements) has been changed. Furthermore, $|\text{val}(up_j) - \text{val}(pr_j)|$ denotes the degree to which the original value of the (product) utility constraint has been changed. Now the Minos solver [Fourer et al., 2002] can calculate a solution to a constraint set adaptation problem (see Definition 13).

If we want to restrict the proposed repair actions to the utility constraints $up_i \in UC$ ($uc_i \in UC$), we have to include $\cup uc_i$ ($\cup up_i$) in the set of constraint definitions (*Con*). The following adaptations (repairs) to the original scoring rules (utility constraints) in *UC* represent a constraint set adaptation for our example constraint set adaptation problem.

4 Anomaly Management for Constraint-based Product Configuration Systems

$profit(investmentperiod_{short}) = 4$
 $availability(investmentperiod_{short}) = 9$
 $profit(investmentperiod_{medium}) = 5$
 $availability(investmentperiod_{medium}) = 4$
 $profit(investmentperiod_{long}) = 8$
 $availability(investmentperiod_{long}) = 1$
 $profit(goal_{rainydays}) = 2$
 $availability(goal_{rainydays}) = 5$
 $profit(goal_{growth}) = 6$
 $availability(goal_{growth}) = 4$
 $profit(goal_{speculation}) = 9$
 $availability(goal_{speculation}) = 2$
 $profitshares(balancedfunds) = 5$
 $availabilityshares(balancedfunds) = 4.99$
 $profitshares(bonds) = 2$
 $availabilityshares(bonds) = 6.33$
 $profitshares(bonds2) = 2$
 $availabilityshares(bonds2) = 7$
 $profitfluctuation(balancedfunds) = 5$
 $availabilityfluctuation(balancedfunds) = 5$
 $profitfluctuation(bonds) = 5$
 $availabilityfluctuation(bonds) = 6.22$
 $profitfluctuation(bonds2) = 6$
 $availabilityfluctuation(bonds2) = 4.67$

The application of these repairs results in the new rankings depicted in Table 4.2. These rankings are now consistent with the set E .

Evaluation

Experiences from Commercial Projects. On the basis of our experiences from commercial recommender projects (see, e.g., Felfernig et al. [2006a]) we identified a clear need for more effective engineering techniques in

4.3 Automated Repair of Scoring Rules

customer	item	utility	ranking after repair	ranking before repair
Robert	balanced funds	160.004	3	1
	bonds	162.004	1	2
	bonds2	161.004	2	3

Table 4.2: Utilities of products for customer Robert (before and after the repair process). The utilities are now consistent with the examples shown in Table 3.16.

the context of utility constraint development and maintenance. The investment recommender of an Austrian financial service provider (see Figure 4.2) has been implemented without the repair functionalities presented in this Section. The system comprises 15 parameters for specifying customer requirements, 10 item variables and about 150 scoring rules (interest dimensions: availability, profit, risk). The recommender application has been designed, developed, and deployed with an overall effort of about 12 man months. Before deploying the first version of the application, new versions of the utility constraint set have been released every third week and tested by domain experts. About 15 adaptation cycles were needed before deploying the utility constraint set in the productive environment. Adaptation efforts related to the utility constraint set consumed about 12 hours per adaptation cycle. This results in 180 hours of development and maintenance efforts specifically related to the adaptation of the utility constraint set. In each adaptation cycle the knowledge engineer tried to adapt the current utility constraint set to be consistent with the example rankings provided by domain experts. The process was error-prone and time-consuming and triggered requirements to automate the adaptation process. The major problem was the task of manually detecting a set of repair actions that make a utility constraint set consistent with the set of examples. Exploiting the presented repair functionalities, a reduction of the overall development and maintenance efforts related to utility constraint sets by about 60% (effort directly related to the adaptation of the utility constraints) can be expected which means more than 100 hours of time savings in projects similar to the described case.

4 Anomaly Management for Constraint-based Product Configuration Systems

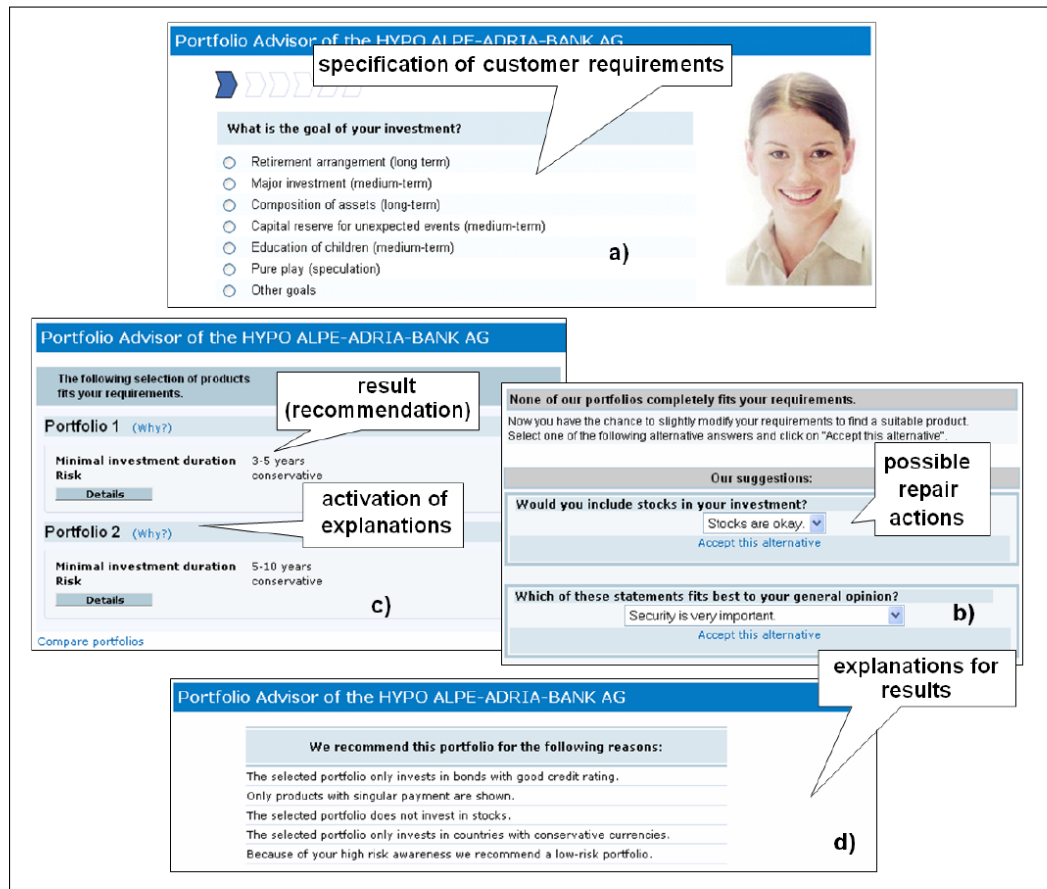


Figure 4.2: Example financial service recommender applications: users (customers and sales representatives) can specify requirements (a). In the case that no solution can be found by the recommender application, possible repair alternatives are presented (b). Solutions (results) are presented in the form of a recommendation list (c) where each entry in the recommendation list is associated with a set of explanations as to why this item has been recommended (d).

Optimality and Performance. Optimality properties of solutions to optimization problems are depending on the used optimization approach [Neumaier et al., 2005]. We had to deal with a non-linear optimization problem since non-linear constraints are part of the constraint set adaptation problem. Non-linear optimization solvers can not guarantee the optimality of an identified solution [Neumaier et al., 2005]. For this reason we had to

4.3 Automated Repair of Scoring Rules

evaluate the quality (degree of optimality) of results calculated by the Minos Solver [Fourer et al., 2002] which we used for calculating solutions to a constraint set adaptation problem. For our evaluations we used commercial utility constraint sets from the product domains of refrigerators (refrig14), financial services (finserv1-4), and computer monitors (mon1-4) (see Table 4.3). For each of the above mentioned application domains we have defined four different settings which differ in the number of examples (#e) and the number of products (#p). For example, in finserv4 #e=20 examples were defined for #p=71 products. The corresponding utility constraint set comprised #su=503 constraints (scoring rules). In order to make the 20 examples consistent with the given set of scoring rules, #so=340 rules have been adapted with an average change distance $\text{avg}(d)=0.056$ where each scoring rule is defined over the domain [0..10]. The time needed by the Minos solver [Fourer et al., 2002] to calculate the adaptations for finserv4 was $t=9464$ milliseconds. The Minos solver is capable of calculating adaptations for faulty utility constraint sets within a reasonable time span acceptable for utility constraints engineering scenarios. In such scenarios the system uses either examples defined by marketing and sales experts or examples automatically derived from existing user interaction logs. In our test settings we used examples which have been specified manually, the automated derivation of examples is the goal for future work. For a detailed discussion of the handling of constraint set adaptation problems without a corresponding solution (e.g., the set of provided examples is inconsistent) the reader is referred to the work of Felfernig et al. [2008b] and Junker [2004].

If repair actions should be done by users of configuration systems, we have to decide if conflicts or diagnoses should be presented to end-users. In the next section we give insights into advantages and disadvantages when conflicts and / or diagnoses and / or utility functions will be presented to end-users.

4 Anomaly Management for Constraint-based Product Configuration Systems

rec.	#e	#p	#su	#so	avg(<i>d</i>)	t(msec)
refrig1	5	16	39	39	< 0.001	761
refrig2	10	30	69	55	0.085	1221
refrig3	15	43	80	62	0.082	1998
refrig4	20	55	100	74	0.077	2252
finserv1	5	19	266	160	0.070	1622
finserv2	10	37	396	244	0.054	5599
finserv3	15	53	439	288	0.055	6389
finserv4	20	71	503	340	0.056	9464
mon1	5	22	109	40	0.046	731
mon2	10	42	177	75	0.041	1813
mon3	15	61	214	105	0.044	1540
mon4	20	80	246	125	0.047	2063

Table 4.3: Performance of the Minos solver [Fourer et al., 2002] for the calculation of solutions for our example test settings in three different domains (refrigerators, financial services, and computer monitors) where #e = no. of examples, #p = no. of products, #su = no. of scoring rules (utility constraints), #so = no. of scoring rules adapted by the non-linear optimization process, avg(*d*) = average distance to the original scoring values (before the repair process has been started), t(msec) = calculation time in milliseconds.

4.4 Visualization of Conflicts⁸

In Section 4.1 we described inconsistencies in terms of conflicts and diagnoses. How users of knowledge-based recommendation systems deal with conflicts, diagnoses, and fitness values will be evaluated in this section. Therefore, we first describe the term *fitness value*. Thereafter, we present an online notebook recommendation system, define hypotheses, and evaluate and discuss them based on an empirical study.

⁸This Section is based on Wotawa et al. [2015b].

Fitness function

We are able to evaluate similarities between products in C_P , the customer preferences in C_R , and the knowledge base C_{KB} . If the customer preferences can not be fulfilled, we can either calculate conflicts and diagnoses or calculate the similarity by using the fitness function given in Equation 4.2.

$$fit(p, C_R) = \sum_{c \in C_R} u(p, c) \times \omega(max_{relevance}, c) \quad (4.2)$$

In Equation 4.2, p defines a product in C_P . C_R is the set of customer preferences. For each customer preference we calculate the utility value $u(p, c)$ and the weighting $\omega(max_{relevance}, c)$. For the utility value we are using McSherry's similarity metrics for each variable [McSherry, 2003]. For example, a lower price value is better (less is better) $\frac{customer_value}{product_value}$, a higher RAM value is better (more is better) $\frac{product_value}{customer_value}$, and for the optical drive a nearer value is better (nearer is better) $= [0, 1]$. The weighting function $\omega(max_{relevance}, c)$ evaluates a weighting for the constraint c by calculating the relative importance $\frac{relevance(c)}{max_{relevance}}$. Note that the application upgraded all fitness values to a percentile value. The best value was the number of fulfilled preferences divided by the number of all the user's preferences.

Notebook Recommendation System

In the preferences screen of our notebook recommendation system (see Figure 4.3) the user is asked for at least three preferences which are described in terms of product variables. Each of the specified preferences must be weighted on a six-point scale.

4 Anomaly Management for Constraint-based Product Configuration Systems

The screenshot shows a web interface titled "Notebook Recommender" with a yellow background. Below the title, it says "Please define at least three preferences and consider that each relevance can not be selected twice!". The main area contains a table with columns for "Preference", "Relevance", and "low" to "high". The "Relevance" column has a blue link. The "low" to "high" columns have radio buttons numbered 1 to 6. The preferences and their selected relevance levels are:

Preference	low	1	2	3	4	5	high
How many cores should your notebook have? CPU = Quad-core	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Which optical drive should your notebook have? Optical Drive = Blu-ray	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
How much should the notebook cost? Price < 750.0 EUR	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
What should be the screen size of your notebook? Screen size = 15.6 Inch	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
What should be the memory capacity of your notebook? RAM > 8 GB	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
What should be the hard disc drive capacity of your notebook? HDD > 320 GB	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

At the bottom of the table is a "start product search" button.

Figure 4.3: Notebook recommendation: definition and weighting of user preferences. Each relevance can only be selected once.

The next step was to remove all products $c \in C_P$ which are consistent with the user preferences C_R to assure that the participants were confronted with a situation where her preferences were inconsistent with the underlying product assortment, i.e., C_R is inconsistent with C_P .

In the following, participants received a visualization of the conflict. Each participant was assigned to one of four scenarios (see Table 4.4). In the first scenario the participants got *minimal diagnoses* as change recommendations (see Figure 4.4). Scenario 2 presents *minimal conflicts* to the participants (see Figure 4.5). Scenario 3 contains both, *minimal diagnoses and minimal conflicts*, as explanations (see Figure 4.6). Scenario 4 shows the fitness values for all products (see Figure 4.7). For the differentiation between experts and novices we used two questions in the questionnaire at the end of the study. The first question asked for a self-assessment and the second question asked for expert knowledge. In our study 111 participants are experts and 90 participants are novices.

Next, we try to find the best approach for presenting inconsistencies in constraint-based recommendation systems. For the evaluation we have measured three general characteristics: a) the *time* which is used to repair an

4.4 Visualization of Conflicts

	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Step 1	Insert preferences			
Step 2	apply diagnoses	dissolve conflicts	apply diagnoses	
Step 3	select a product			
Step 4	answer a questionnaire (2 pages)			

Table 4.4: Overview of the user activities and scenarios

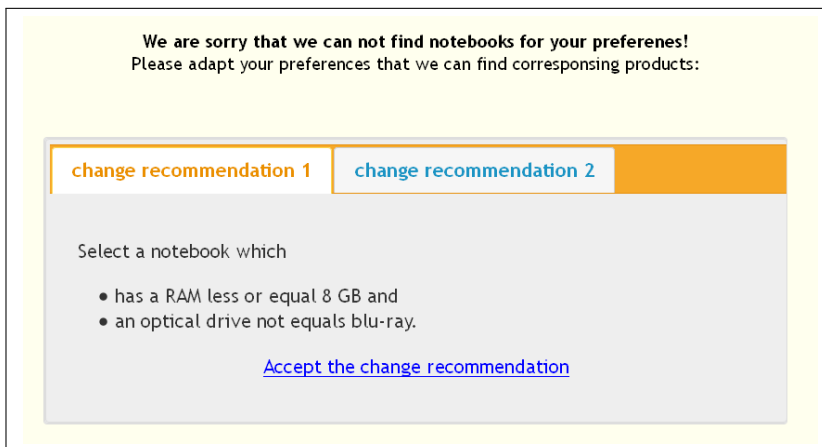


Figure 4.4: Presentation of 1 to n diagnoses denoted as 'change recommendation'

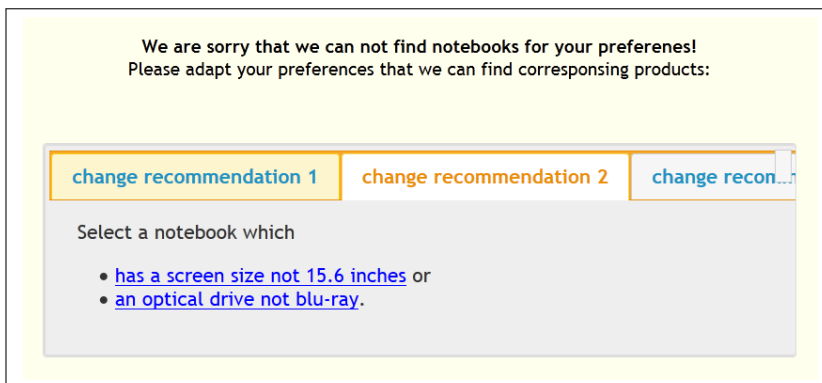


Figure 4.5: Presentation of 1 to n conflicts. Each 'change recommendation' is a conflict set. The user has to remove one constraint from any conflict set.

4 Anomaly Management for Constraint-based Product Configuration Systems

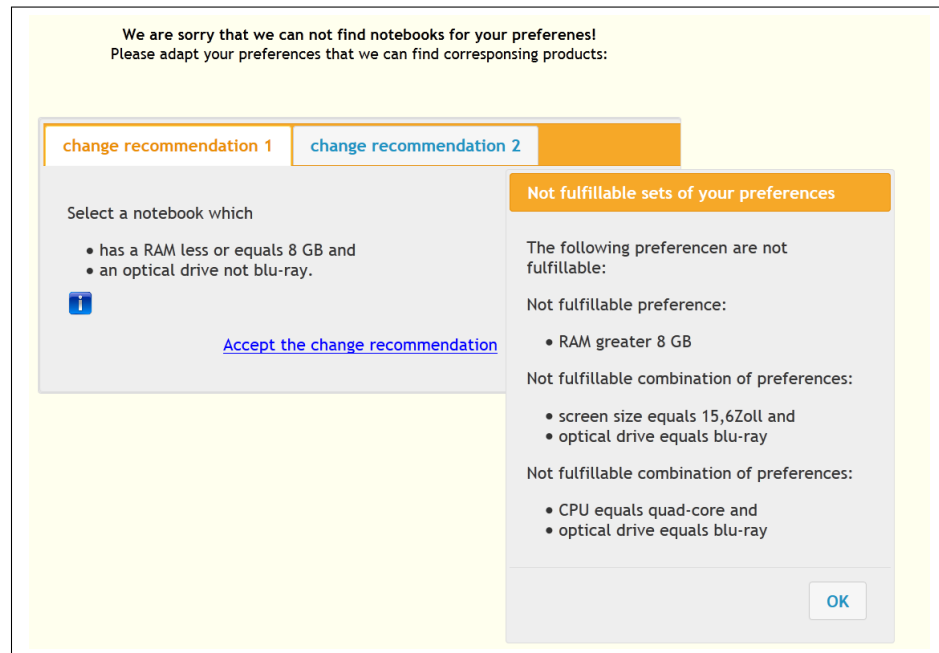


Figure 4.6: Presentation of 1 to n diagnoses and conflicts. The user has to apply one diagnosis ('change recommendation'). The diagnoses are explained by the underlying conflict sets.

We are sorry that we can not find notebooks for your preferences!

The following products are the most similar ones compared to your preferences:

Name	RAM	LCD size	Optical Drive	HDD	Price	CPU	Accordance	
Samsung 550P7C	8.0 GB	17.3 inch	DVD	1000.0 GB	856.0 EUR	Quad-core	83%	select
ASUS N56VB-S4042H (90NB0161-M00520)	8.0 GB	15.6 inch	DVD	1000.0 GB	949.0 EUR	Quad-core	83%	select
Acer Aspire V3-771G-736b8G75Maii	8.0 GB	17.3 inch	DVD	750.0 GB	776.0 EUR	Quad-core	79%	select
HP Pavilion g6-2207sg	8.0 GB	15.6 inch	DVD	750.0 GB	599.0 EUR	Quad-core	78%	select
MSI GE70-i760M285E (001756-SKU11)	8.0 GB	17.3 inch	DVD	500.0 GB	908.0 EUR	Quad-core	74%	select
Lenovo G580	4.0 GB	15.6 inch	DVD	1000.0 GB	589.0 EUR	Quad-core	74%	select
Lenovo IdeaPad Z585	4.0 GB	15.6 inch	DVD	750.0 GB	527.0 EUR	Quad-core	69%	select
MSI GE70-i760M245 (001756-SKU10)	4.0 GB	17.3 inch	DVD	500.0 GB	978.0 EUR	Quad-core	65%	select
Medion Akoya E7221	8.0 GB	17.3 inch	DVD	1000.0 GB	579.0 EUR	Dual-core	65%	select
HP Pavilion g6-2222sg	4.0 GB	15.6 inch	DVD	500.0 GB	553.0 EUR	Quad-core	64%	select

Not fulfilled preference:
• Price < 750.0 EUR

Figure 4.7: Presentation of fitness values

inconsistency, b) the *understandability* of conflicts and diagnoses, and c) the *satisfaction* with the support for the 'no solution could be found' dilemma.

Hypotheses

Having selected a diagnosis (in Scenarios 1 and 3), the participant (user) receives a list of notebooks. In Scenario 2 the user a) removed at least one of her preferences from each conflict set or b) at least one conflict set still contains all constraints and the participant additionally has to remove another preference. We call the number of preferences, which are removed until the user receives products, interaction cycles. For example, an interaction cycle of two means that the user removed two of her preferences until products could be presented. Therefore we expect that the time, which is necessary for resolving the conflict, will be lower when diagnoses are presented to the participant:

Hypothesis 1: Study participants will solve inconsistencies faster when they receive diagnoses.

The study participants received all diagnoses in a preferred order. We expect that the first diagnosis will be selected most frequently.

Hypothesis 2: The first diagnosis will be selected by the majority of the users for adapting their preferences.

A conflict occurs if a set of preferences can not be fulfilled (see Definitions 4 and 5). Scenario 3 uses the minimal conflict sets (see Definition 5) as a description for the minimal diagnoses (see Definition 7). We expect a positive impact on the understandability by the diagnoses:

Hypothesis 3: Participants will understand their conflicts more easily if they receive explanations.

4 Anomaly Management for Constraint-based Product Configuration Systems

When the participants do not receive products after having inserted the preferences, the satisfaction with the recommendation system will decrease and we expect that the satisfaction with the product assortment of our recommendation system will be higher if products are offered (Scenario 4), even if they do not fulfill all of the participants' preferences.

Hypothesis 4: The participants will have a higher satisfaction with the product assortment when they receive fitness values (Scenario 4, see Figure 4.7).

Due to the stability of preferences [Tversky et al., 1990], the participants are less willing to adapt their preferences. When the recommendation system asks for more than one adaption of preferences, the participants will have a lower satisfaction with the system. This leads to the following hypothesis:

Hypothesis 5: More interaction cycles lead to a lower satisfaction with the anomaly support.

Evaluation

For evaluating our hypotheses, we conducted a study at the Graz University of Technology and the University of Klagenfurt. 240 users participated in our study. The students' average age is 25 years (std. dev.: 5.52 years). The participants are studying technical sciences (117), cultural sciences (63), economics (29), and other sciences (n = 31). We have tested our results with a two-tailed Mann-Whitney U-test and removed all participants with contradictory answers to the SUS (system usability scale) questionnaire [Brooke, 1986]. Finally, we divided 201 participations into the scenarios with diagnoses (n = 56), conflicts (n = 50), diagnoses and conflicts (n = 38), and fitness function (n = 57).

4.4 Visualization of Conflicts

Hypothesis 1 focuses on the time which is required to resolve inconsistencies. Therefore, we measured the time between the first conflict notification and the product presentation (see Table 4.5).

Scenario	1 D	2 C	3 D&C	4 Fit
conflict solving time	16.64	21.16	20.05	0.00
product selection time	26.09	27.52	18.72	43.82
total	42.73	48.68	38.77	43.82

Table 4.5: Average time (in sec.) to resolve inconsistencies and to select a product (in sec.; D = diagnoses, C = conflicts, Fit = fitness)

The result shows that the time for removing conflicts with diagnoses is lower (16.64 sec.) than with conflicts (21.16 sec.) or selecting the diagnoses with a corresponding explanation (20.05 sec.). This is because there is only one interaction cycle for resolving inconsistencies with a diagnosis whereas the average number of interaction cycles to resolve inconsistencies with conflicts is 1.66. Reading the explanation of a diagnosis also increased the time to resolve an inconsistency (20.05 sec.), compared to the diagnoses without explanations ($p < 0.1$). The time for resolving the conflicts is 0 in Scenario 4 since they are not resolved. Since these results are statistically significant, **these results confirm Hypothesis 1**.

The time to select a product was nearly the same in the scenarios with diagnoses (Scenario 1) and conflicts (Scenario 2). The third scenario performs best in terms of the time which is required to select a product (18.72 sec.). This can be explained by the fact that dealing with diagnoses and conflicts helps to receive a deep understanding of the problem. Participants in the fourth scenario required 43.82 sec. for selecting a product. The higher effort for selecting a product can be explained by the missing explanations of the conflict, and the participants may get confused that not all preferences are fulfilled by the offered products. All differences in the product selection time are statistically significant ($p < 0.001$).

4 Anomaly Management for Constraint-based Product Configuration Systems

We also researched the influences of the number of conflicts and diagnoses (see Table 4.6).

# of presented diagnoses / conflicts	n	satisfaction	repair time
1 diagnosis:	11	4.55	11.18 sec.
2 diagnoses:	11	4.14	10.71 sec.
> 2 diagnoses:	38	4.37	19.32 sec.
1 conflict:	56	4.09	22.29 sec.
2 conflicts:	23	4.04	45.48 sec.
>2 conflicts:	4	1.75	62.00 sec.

Table 4.6: Average time to repair inconsistencies regarded to the number of presented diagnoses and conflict sets (1 represents the lowest and 5 represents the highest possible value)

The number of diagnoses presented to the participant does not have an impact on the satisfaction of the anomaly presentation. If a participant receives more than two diagnoses, the time for applying a diagnosis is nearly two times higher. On the other hand, the satisfaction with the anomaly presentation decreases enormously if a participant has to remove more than two conflicts. Also the time to remove one constraint from a conflict set is much higher than to apply a diagnosis. The decreasing satisfaction can be explained by the user interaction. While applying a diagnosis (one interaction cycle) resolves all conflict sets, many conflict sets lead to many different repair actions (many interaction cycles) and we expect that especially more than two interaction cycles lead to a high participants' frustration. All values are not statistically significant ($p > 0.1$).

Hypothesis 2 is looking at the ordering of preferred diagnoses and conflicts. Therefore we measured the position of the selected conflict / diagnosis (see Figure 4.8). Note, there are only those participants considered out of Scenarios 1 and 3 whose number of offered diagnoses is greater than one.

4.4 Visualization of Conflicts

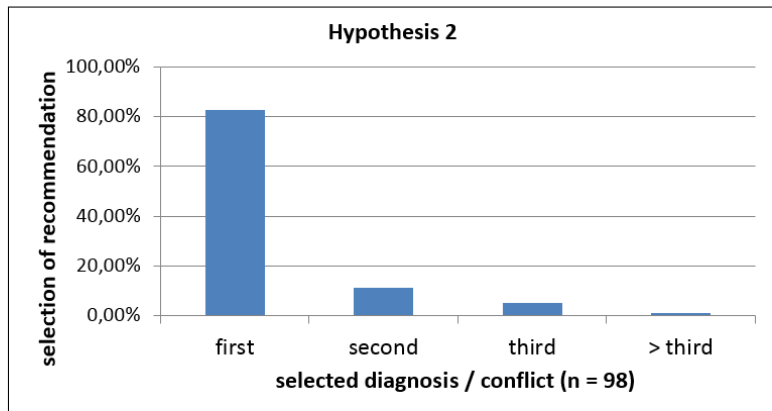


Figure 4.8: Ranking of selected diagnosis / conflict

We can confirm **Hypothesis 2** since 81 of the participants (82.65%) selected the first diagnosis. The second diagnosis was selected by 11 (11.22%), the third one by 5 (5.10%) participants and the fourth recommendation by one participant (1.02%). Reasons for applying the first diagnosis are that the first diagnosis contains only unimportant preferences⁹, the primacy-effect [Felfernig et al., 2007a], and preference reversals [Tversky et al., 1990].

For measuring **Hypothesis 3** we asked the participants of the Scenarios 1-3 if the diagnoses/conflicts were understandable. Answers were given on a 5 point Likert-scale (5 represents the highest understandability).

Results show that the highest understandability is given when diagnoses are presented (Scenario 1) followed by diagnoses explained with conflicts (Scenario 3) and conflicts (Scenario 2, see Table 4.7). The difference between the understandability of conflicts (4.40, Scenario 2) and the other scenarios (Scenario 1 with 4.55 and Scenario 3 with 4.45) is statistically significant ($p < 0.05$). The degree of understandability is higher for experts than for novices ($p > 0.1$). We can **partially confirm Hypothesis 3** since experts have a higher understanding of the conflict, when conflicts and diagnoses are

⁹The preferences are initially ranked by all participants. Since the ranking is based on a six point scale, we aggregated the values of those preferences that are in a diagnosis.

4 Anomaly Management for Constraint-based Product Configuration Systems

presented, whereas novices can not deal with much information. Due to the cognitive processes (trial-and-error of novices versus analytical processing of experts [Hong and Liu, 2003]), it is easier to deal with diagnoses when the cognitive process is more analytical. When participants use a trial-and-error process and they do not expect the visualization of conflicts, it is harder to adapt the preferences.

Scenario	1 D	2 C	3 D&C
Total:	4.55	4.40	4.45
Experts:	4.62	4.38	4.67
Novices:	4.46	4.42	4.18

Table 4.7: Understandability of inconsistencies

Hypothesis 4 evaluates the satisfaction with the recommended products. The average values are from 2.62 up to 3.3 (see Table 4.8) which is worse and can be explained by the removal of all products that are conform to the participants' preferences at the beginning of the process.

Scenario	1 D	2 C	3 D&C	4 Fit
Total	2.62	3.30	2.80	3.30
Experts	2.44	3.12	2.33	3.19
Novices	2.88	3.50	3.35	3.48

Table 4.8: Satisfaction with the product assortment

The results show that conflicts (Scenario 2) and the fitness function (Scenario 4) lead to the highest satisfaction with the product assortment. A differentiation between experts and novices does not influence the significance. As conflicts and the fitness values lead to the same satisfaction, **we can not confirm Hypothesis 4**. An interesting result is that novices, compared to experts, have an overall higher satisfaction with the product assortment. This can be explained by the fact that they are more happy to

4.4 Visualization of Conflicts

get any products recommended. On the other hand, experts know that there are products which fit with their preferences.

Hypothesis 5 will be evaluated by Table 4.9. There is a significant difference when participants had more than two interaction cycles. A statistically significant difference between experts and novices is not constituted. A differentiation between the interaction cycles of diagnoses and conflicts also does not lead to a significant difference between all interaction cycles or between conflict and diagnoses visualization. Since a higher satisfaction is given if participants have only one or two interaction cycles, **we can confirm Hypothesis 5**.

# interaction cycles	n	satisfaction
1	34	4.44
2	10	4.30
3	3	2.67
≥ 4	3	3.00

Table 4.9: Satisfaction with the presented conflicts regarding interaction cycles

A comparison between the number of conflicts / diagnoses and satisfaction, understandability, or time to resolve the inconsistency is not statistically significant.

Discussion

This section gave an overview of conflict visualization in constraint-based recommendation systems. If we can not present products that fit to the user's preferences, the user has to adapt her preferences. Such preference reversals always result in a low satisfaction of users. The degree of dissatisfaction depends on how often the preferences have been fulfilled in the past [Tversky et al., 1990].

4 Anomaly Management for Constraint-based Product Configuration Systems

If users have positive experience with their preferences, it can happen that the participants have well-established anchoring affects [Tversky and Kahneman, 1974]. In such scenarios the participants may have stable preferences but preference reversals are necessary to get notebooks. It can be more problematic if there are many conflicts / diagnoses shown because it could be the case that a representation of all conflicts / diagnoses leads to a manifestation of the current preferences and the user is less willing to accept any conflicts / diagnoses. Such an effect is called status-quo bias [Kahneman et al., 1991; Samuelson and Zeckhauser, 1988].

Another important aspect is the cognitive processing task. While novices tend to use trial-and-error processes, experts tend to use heuristic and analytic cognitive processes [Hong and Liu, 2003]. That means that novices tend to adapt their preferences unless they receive products. Our results confirm the usage of the trial-and-error process since the satisfaction of novices is high if they can adjust their preferences arbitrarily or receive similar products (see Hypothesis 4). On the other hand, experts try to understand the modifications and analyze them. Therefore, they prefer the visualization of diagnoses (see Hypothesis 3).

This chapter described how we can support knowledge engineers and end-users when they have to deal with different types of anomalies. Besides anomalies there are also several techniques to reduce the complexity of a constraint-based knowledge base or to give new insights into the knowledge base. Such techniques will be described in the following chapter.

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

In this chapter we describe supporting techniques for knowledge engineers who have to maintain a constraint-based product configuration knowledge base. First, we show how we can divide the information of a knowledge base into relevant and irrelevant information (Section 5.1). Since anomaly management is a critical task when maintaining product configuration knowledge bases, Section 5.2 gives an overview of algorithms for detecting conflicts, diagnoses, redundancies, and well-formedness violations based on constraints and assignments. Detecting dependencies between variables and constraints is a crucial task for the maintenance. Therefore, Section 5.3 gives an overview of dependency detection and introduces a simulation technology for approximating the consistency rate of a knowledge base. If a knowledge engineer wants to know the complexity of a knowledge base, it is possible to approximate the complexity with our evaluation techniques in Section 5.4. Finally, we also take a look at automated test case generation (Section 5.5) and the development of knowledge bases (Section 5.6) as supporting techniques for the development and maintenance of product configuration systems.

5.1 Recommendation Techniques¹

Constraint-based knowledge bases can have hundreds or thousands of variables, domain elements, and constraints. If there is a maintenance task (e.g., inserting new tire sizes), recommendation techniques help to differentiate between relevant and not relevant information within the knowledge base related to the maintenance task. For example, the tires of a bike probably have an influence on the frame of a bike but do not influence the bell of a bike. In such cases recommendation techniques detect items (variables, domain elements, constraints, test cases) which are influenced by the tires and the knowledge engineer can focus on these items. We describe four different types of recommendation to support knowledge engineers in their maintenance tasks [Felfernig et al., 2013b].

The first recommendation approach is the *most viewed* recommendation which is user-independent. It can be useful for new engineers of a product domain.

Second, *recently added* lists new items (products, product variables, questions, and constraints) in the knowledge base. It is user-dependent since it considers the knowledge engineer's last log in and helps to get a fast understanding of the previous changes in the knowledge base.

The next type of recommendation is *collaborative filtering*. This type of recommendation takes the ratings for items into account and looks for knowledge engineers with similar ratings. In our case we do not have ratings but we use the interaction with items as ratings. If a knowledge engineer looks at products, she 'rates' the item with 1. 2 will be added by the knowledge engineer if she is editing an item. Table 5.1 shows an example for a collaborative filtering recommendation for knowledge engineer u_0 based on our example 3.2.

¹This Section is based on Felfernig et al. [2013b].

5.1 Recommendation Techniques

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
u_0		1		1	2	1		?	
u_1		1	1		1	1		1	
u_2		1	2	1			2	2	
u_3	1		1				2		

Table 5.1: An example for collaborative filtering. 1 means that the item c_i is viewed by the user u_j , 2 means that the item is edited and ' ' means that the item is neither viewed nor edited by the user.

In Table 5.1 we try to find out if we should recommend item c_7 to knowledge engineer u_0 . The common process to find recommendable items is twofold. First, we try to find knowledge engineers with similar interests. In our example u_1 and u_2 have nearly the same items viewed or edited. Second, we have to evaluate if the similar knowledge engineers are interested in the item. Therefore, we use the Pearson correlation [Felfernig et al., 2014a; Jannach et al., 2010]. In our example u_1 , and u_2 have viewed / edited item c_7 and we should recommend c_7 to knowledge engineer u_0 .

Another recommendation approach is the usage of *content-based filtering*. The basic idea is to find similar items compared to a reference item. We take the variable names and domain values from a constraint and evaluate the similarities between the reference item and all other items. The similarities are measured by the TF-IDF (term frequency and inverse document frequency) algorithm [Jannach et al., 2010; Pazzani and Billsus, 2007] where the item is the document and the terms are the variables and domain elements. Table 5.2 shows the similarity values between constraint c_7 as reference constraint with the other constraints.

With this approach we can say that there is a high relationship between constraint c_7 with the constraints c_0 and c_2 and a weak relationship with the constraints c_6 and c_1 .

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

constraint	similarity
c_0	0.50
c_1	0.17
c_2	0.50
c_3	0.00
c_4	0.00
c_5	0.00
c_6	0.33
c_8	0.00

Table 5.2: Similarities between constraint c_7 with the other constraints based on content based recommendation

This section described the relationship of elements within a knowledge base based on the relationship of elements (e.g., the variables within a constraint) or knowledge engineers. Another relationship can be based on anomalies (e.g., two constraints are within a conflict set). How we can calculate anomalies efficiently will be described in the next section.

5.2 Anomaly Management²

Anomalies are patterns in data that do not conform to a well defined notion of normal behavior [Chandola et al., 2009]. We describe and evaluate the detection of inconsistencies (Section 4.1.1). Thereafter, we present algorithms for the detection of redundancies (Section 4.1.2) and introduce an example for assignment-based redundancy detection. Finally, we also present algorithms for the detection of well-formedness violations (Section 4.1.3) to support knowledge engineers in terms of increasing the understandability of knowledge bases in this section.

²This Section is based on Reinfrank et al. [2015a]; Felfernig et al. [2012a]; Schubert et al. [2011].

5.2.1 Inconsistencies

Inconsistencies are probably the most discussed type of anomaly. A knowledge base is inconsistent if there does not exist a consistent and complete solution (i.e., a valid instance, see Definition 3) to the constraint satisfaction problem (CSP) defined by the knowledge base. In such situations we have to identify at least one conflict set which is a set of constraints of the knowledge base.

If an inconsistency exists in the configuration knowledge base, we are able to identify all conflict sets $CS \subseteq C_{KB}$ [Felfernig and Schubert, 2011b] which do not allow the determination of a solution. A conflict set CS is minimal if there does not exist a conflict set CS' with $CS' \subset CS$ (see Definition 5). The standard algorithm for detecting a minimal conflict set is QuickXPlain [Junker, 2004]. QuickXPlain is based on the idea of divide-and-conquer where the basic strategy of the algorithm is to filter out constraints - which are not part of a minimal conflict - as soon as possible (see Algorithms 1 and 2).

The following algorithms have two sets as parameters which are representing the knowledge base: the set C_{KB} is a set of constraints which can not be part of a conflict set because, for example, they have already been tested by the knowledge engineer. C_R is a set of constraints which can contain a conflict set because, for example, errors during the last update are inserted. The union of both sets $C_{KB} \cup C_R$ represents all constraints $C = C_{KB} \cup C_R$.

Algorithm 1 QUICKXPLAIN(C_{KB}, C_R): Δ

▷ $C : KB$: set of not diagnosable constraints
 ▷ C_R : set of diagnosed constraints

if *isEmpty*(C) or *consistent*($C_{KB} \cup C_R$) **then**
 return \emptyset ;
else
 return *QuickXPlain'*(C_{KB}, C_{KB}, C_R);
end if

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

If $C_{KB} \cup C_R$ is inconsistent and $C_R \neq \emptyset$, the preconditions of applying QuickXPlain are fulfilled and Algorithm 2 will detect a minimal conflict. This algorithm divides C_R into C_1 and C_2 and checks whether $C_{KB} \cup C_1$ is inconsistent. If it is inconsistent, an empty set will be returned, otherwise C_1 will be divided and tested again. If $\text{singleton}(C)$ is true, constraint $c \in C$ will be part of the minimal conflict set and inserted into Δ_1 . Δ_2 receives a constraint set which will also be a part of the minimal conflict and can be \emptyset or $\Delta_2 \subseteq C_1$. All constraints in Δ_1 and Δ_2 are part of the minimal conflict set and returned by the algorithm. $\Delta_1 \cup \Delta_2 = \emptyset$ means that all conflict sets are in C_{KB} and the knowledge base can not be consistent without removing constraints from C_{KB} .

Algorithm 2 QUICKXPLAIN'($C_{KB}, \Delta, C_R = \{C_1, \dots, C_r\}$): Δ

▷ C_{KB} : Set of diagnosed constraints which are not part of a diagnosis
 ▷ Δ : Set of diagnosed constraints which are part of a diagnosis
 ▷ C_R : Set of constraints which will be diagnosed

if $\Delta \neq \emptyset$ **and** $\text{inconsistent}(C_{KB})$ **then**
 $\text{return } \emptyset$;
end if
if $\text{singleton}(C_R)$ **then**
 $\text{return } C_R$;
end if
 $k \leftarrow \lceil \frac{r}{2} \rceil$;
 $C_1 \leftarrow \{c_1, \dots, c_k\}$;
 $C_2 \leftarrow \{c_{k+1}, \dots, c_r\}$;
 $\Delta_1 \leftarrow \text{QuickXPlain}'(C_{KB} \cup C_1, C_1, C_2)$;
 $\Delta_2 \leftarrow \text{QuickXPlain}'(C_{KB} \cup \Delta_1, \Delta_1, C_1)$;
 $\text{return } (\Delta_1 \cup \Delta_2)$;

Where minimal conflict sets represent minimal sets of constraints which do not allow the calculation of a solution, diagnoses are minimal sets of constraints which have to be deleted from the knowledge base such that a solution can be identified for the remaining set of constraints. More formally, a diagnosis $\Delta \subseteq C_R$ is a set of constraints such that $C_R \setminus \Delta$ is consistent, i.e., there exists at least one solution for $C_R \setminus \Delta$. A diagnosis Δ is minimal if there does not exist a diagnosis Δ' with $\Delta' \subset \Delta$.

In contrast to QuickXPlain the FastDiag algorithm [Felfernig et al., 2012b] (see Algorithms 3 and 4) calculates one minimal diagnosis $\Delta \subset C_R$ s.t. $C_R \setminus \Delta$ is consistent. Similar to QuickXPlain [Junker, 2004] FastDiag [Felfernig et al., 2012b] exploits a divide-and-conquer strategy. However, the focus is different: it determines a minimal diagnosis as opposed to a minimal conflict set determined by QuickXPlain.

Similar to QuickXPlain Algorithm 3 receives a set C_R which contains all diagnosable constraints. Contrary to QuickXPlain Junker [2004], the second set is denoted as C and contains all constraints. If C_R is an empty set, FastDiag has no diagnosable set and the algorithm skips all further steps. It also stops if the set $C \setminus C_R$ is inconsistent because this set contains inconsistencies but it will not be diagnosed. If both preconditions are fulfilled, Algorithm 4 will diagnose C .

Algorithm 3 FASTDIAG(C_R, C): Δ

▷ C_R : Set of constraints which will be diagnosed

▷ C : inconsistent configuration knowledge base including all constraints

if $C_R = \emptyset \vee inconsistent(C \setminus C_R)$ **then**
 return \emptyset ;
else
 return $DIAG(\emptyset, C_R, C)$
end if

First of all, DIAG checks whether C is consistent. If it is consistent, each subset of C is also consistent and no constraint in C will be a part of the diagnosis. C_R will be divided into two subsets C_1 and C_2 . Each subset will be removed from C separately and within a recursion which means that the subsets will be further divided if an inconsistency is still given. If $C - C_1$ is consistent, we can say that C_2 is consistent and an empty set will be returned. If it is inconsistent, at least one constraint in C_1 must be part of the diagnosis and therefore C_1 will be divided and tested again unless $|C| = 1$. In this case DIAG returns this constraint as a part of the diagnosis. The algorithm returns a set $\Delta_1 \cup \Delta_2$ of constraints which represent a minimal diagnosis.

Algorithm 4 $DIAG(\Delta, C_R = \{c_1, \dots, c_r\}, C):\Delta$

```

▷  $\Delta$ : Set of diagnosed constraints which are part of a minimal diagnosis
    ▷  $C_R$ : Set of constraints which will be diagnosed
    ▷  $C$ : Set of diagnosed constraints which are not part of a diagnosis
if  $\Delta \neq \emptyset$  and  $consistent(C)$  then
    return  $\emptyset$ ;
end if
if  $singleton(C_R)$  then
    return  $C_R$ ;
end if
 $k \leftarrow \lceil \frac{r}{2} \rceil$ ;
 $C_1 \leftarrow \{c_1, \dots, c_k\}$ ;
 $C_2 \leftarrow \{c_{k+1}, \dots, c_r\}$ ;
 $\Delta_1 \leftarrow DIAG(C_2, C_1, C \setminus C_2)$ ;
 $\Delta_2 \leftarrow DIAG(\Delta_1, C_2, C \setminus \Delta_1)$ ;
return  $(\Delta_1 \cup \Delta_2)$ ;

```

In order to analyze the performance of QuickXPlain (Algorithm 1) and FASTDIAG (Algorithm 3), we conducted a performance analysis for both algorithms on the basis of different feature models provided by the S.P.L.O.T. repository. The results of this analysis are presented in the following.

Performance evaluation For evaluation purposes we selected different feature models offered by the S.P.L.O.T. repository: *Car Selection* (Table 5.3), *SmartHome V. 2.2.* (Table 5.4), and *Xerox* (Table 5.5). In order to evaluate the performance of FASTDIAG, we randomly inserted additional constraints in the knowledge bases for inducing inconsistencies which could then be exploited for determining minimal diagnoses. For a systematic evaluation we generated different versions of the (inconsistent) feature models that differed in terms of their inconsistency rate (see Formula 5.1) which was categorized in $\{2\%, 5\%, 7\%\}$. We used a random variable to control the degree of generated inconsistencies (the number of conflicts) in a feature model. As reasoning engine we used the CHOCO constraint solving library.³

³www.emn.fr/z-info/choco-solver

5.2 Anomaly Management

In order to import feature models to our environment, we implemented a parser that generated CHOCO knowledge bases from S.P.L.O.T. SXFM based feature models.

$$\text{Inconsistency Rate} = \frac{\#conflicts\ in\ C}{\#constraints\ in\ C} \quad (5.1)$$

Feature Model: Car Selection						
# Variables: 72						
# Constraints: 96						
#D	Inconsistency Rate					
	2% (8 diagnoses)		5% (64 diagnoses)		7% (182 diagnoses)	
	FastDiag	HSDAG	FastDiag	HSDAG	FastDiag	HSDAG
1	452	874	561	1888	858	5366
2	749	890	920	1891	1638	5382
3	1045	921	1294	2138	2059	5506
4	1373	936	1653	2143	2324	5522
5	1529	968	1872	2262	2464	5544
10	-	-	2511	2418	2932	5709
20	-	-	2964	2450	3806	6162
all	1632	1027	4383	3339	11856	8860

Table 5.3: Evaluation of FastDiag and HSDAG with the Car Selection feature model from S.P.L.O.T. (#D = number of diagnoses).

The performance tests were executed within a Java application running on a 64bit Windows 7 desktop PC using 8GB RAM and an Intel(R) Core(TM) i5-2320 CPU with 3.0GHz. Each run of the diagnosis algorithm for a specific setting has been repeated 10 times were in each run the ordering of the constraints was randomized. For each setting we evaluated the runtime (in ms) of both, the standard hitting set based approach to the termination of diagnoses (HSDAG; Reiter [1987]) and FASTDIAG. As scenario we chose the diagnosis of *void feature models* where we induced different degrees of inconsistency (based on the *inconsistency rate measure* - see Formula 5.1). The

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

Feature Model: SmartHome V. 2.2						
# Variables: 61						
# Constraints: 63						
#D	2% (8 diagnoses)		Inconsistency Rate		7% (77 diagnoses)	
	FastDiag	HSDAG	FastDiag	HSDAG	FastDiag	HSDAG
1	297	920	312	952	577	2683
2	427	967	452	968	951	2684
3	609	983	592	983	1341	2686
4	734	998	733	1139	1762	2699
5	843	1014	842	1155	2090	2671
10	-	-	967	1529	2792	2715
20	-	-	-	-	3369	2746
all	1155	1061	1606	1545	6224	3151

Table 5.4: Evaluation of FastDiag and HSDAG with the SmartHome V 2.2 feature model from S.P.L.O.T. (#D = number of diagnoses).

upper bound for the evaluation time was set to 100.000 ms – in the case that this upper limit was exceeded, the search was stopped.

Next, we used three different set-ups for the performance analysis, differing in the number of diagnoses which are calculated to give a detailed description when conflict detection and HSDAG perform better than FastDiag. Figure 5.1 presents results for calculating 1, 5, and all diagnoses. For the calculation of diagnoses with the QuickXPlain algorithm (Algorithm 1 and 2) and for getting more than one diagnosis from FastDiag (Algorithm 3 and 4) we have used Reiter's HS-tree [Reiter, 1987]. With this tree it is possible to calculate all conflicts and diagnoses for a given configuration knowledge base by transferring each constraint from the result of QuickXPlain / FastDiag separately from the set of diagnosable constraints C_R to the set of not diagnosable constraint set C_{KB} . For a more detailed description of the HS-tree we refer the reader to [Reiter, 1987].

5.2 Anomaly Management

Feature Model: Xerox						
# Variables: 172						
# Constraints: 205						
#D	2% (140 diagnoses)		Inconsistency Rate		7% (55 diagnoses)	
	FastDiag	HSDAG	FastDiag	HSDAG	FastDiag	HSDAG
1	1638	3354	1260	2996	1740	3023
2	2013	6646	1710	3167	2050	3203
3	2262	12106	1970	9454	2330	9544
4	2434	12355	2180	9536	2580	9654
5	2637	28111	2341	12044	2790	12165
10	3417	69950	2921	64631	3330	65240
20	4758	75317	3911	90715	5010	91726
all	46785	>100000	17301	>100000	10541	>100000

Table 5.5: Evaluation of FastDiag and HSDAG with the Xerox feature model from S.P.L.O.T. (#D = number of diagnoses).

Results show that the FastDiag approach has an advantage compared to the QuickXPlain algorithm if one diagnosis has to be calculated. The reason for that is that the calculation of a diagnosis in the HS-tree, when using the FastDiag approach, is done when the first node in the HS-tree is calculated. On the other hand, the conflict set approach must expand a path from the root to a leaf before having a diagnosis. This approach has an advantage if many diagnoses have to be calculated.

Finally, if all diagnoses have to be calculated, QuickXPlain performs better, because the reuse of previously calculated conflict sets within the HS-tree increases and the number of expanded nodes in the HS-tree is lower.

5.2.2 Redundancies

Redundancies are anomalies which do not influence the behavior of the knowledge base. A constraint c_a is redundant if the deletion of the

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

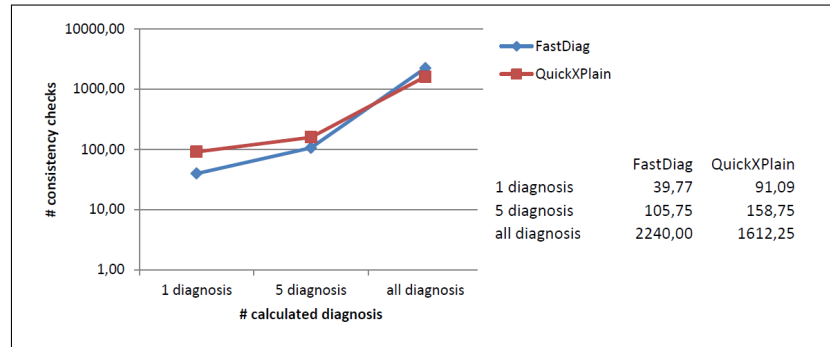


Figure 5.1: Number of consistency checks for calculating diagnoses

constraint will not influence the behavior of the configuration knowledge base, more formally described as $C \setminus \{c_a\} \models c_a$. A constraint c_a is said to be non-redundant if the negation of C (i.e. \bar{C}) is consistent with $C - \{c_a\}$. The redundancy detection algorithms can be applied only if C is consistent and no inconsistencies are in C .

The first approach to the detection of redundancies has been proposed by Piette [Piette, 2008]. The approach is the following: a knowledge base with its negotiation must be inconsistent, formally described as $C \cup \bar{C} = \emptyset$. By removing each constraint c_a separately from C , the algorithm checks whether the result of $C \setminus \{c_a\} \cup \bar{C}$ is still inconsistent. If this is the case, then the constraint is redundant and can be removed.

After each constraint has been checked separately, C_t is a non-redundant constraint set (minimal core) which means that $\Delta = C \setminus C_t$ is a set of redundant constraints in C and these are returned.

An alternative approach (CoreDiag) has been proposed by Felfernig et al. [2011b]. Instead of a linear approach they adapt the QuickXPlain algorithm. The divide-and-conquer approach of this algorithm checks whether removing a set of constraints leads to an inconsistency, formally described as $C \setminus \Delta \cup \bar{C} = \emptyset$. If it is not inconsistent, C must be further divided and

Algorithm 5 SEQUENTIAL(C): Δ

▷ C: configuration knowledge base
 ▷ \bar{C} : the complement of C
 ▷ Δ : set of redundant constraints

```

 $C_t \leftarrow C;$ 
for all  $c_i$  in  $C_t$  do
  if isInconsistent( $C_t - c_i \cup \bar{C}$ ) then
     $C_t \leftarrow C_t - \{c_i\};$ 
  end if
end for
 $\Delta \leftarrow C - C_t;$ 
return  $\Delta;$ 

```

tested again. Similar to SEQUENTIAL the CoreDiag algorithm also has C as input.

Algorithm 6 COREDIAG (C): Δ

▷ $C = \{c_1, c_2, \dots, c_n\}$
 ▷ \bar{C} : the complement of C
 ▷ Δ : set of redundant constraints

```

 $\bar{C} \leftarrow \{\neg c_1 \vee \neg c_2 \vee \dots \vee \neg c_n\};$ 
return( $C \setminus \text{CORED}(\bar{C}, \bar{C}, C)$ );

```

CoreD (see Algorithm 7) checks if $C_{KB} \subseteq C_R$ is inconsistent. An inconsistency of $C_{KB} \cup \bar{C}_R$ means that the subset is not redundant and no constraint of C_{KB} will be a part of Δ . *singleton*(C_R) is true means that this constraint is part of the diagnosis and will be returned. Otherwise the constraint set C_R will be further divided and the subsets will be checked recursively.

Which algorithm should be used and which preconditions influence the selection of the algorithm, will be described in an empirical study in the next Section.

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

Algorithm 7 CORED($C_{KB}, \Delta, C_R = \{c_1, c_2, \dots, c_r\}$): Δ

▷ C_{KB} : Set of diagnosed constraints which are not part of a diagnosis
 ▷ Δ : Set of diagnosed constraints which are part of a diagnosis
 ▷ C_R : set of constraints to be checked for redundancy

if $\Delta \neq \emptyset$ **and** *inconsistent*(C_{KB}) **then**
 return \emptyset ;
end if

if *singleton*(C_R) **then**
 return(C_R);
end if

$k \leftarrow \lceil \frac{r}{2} \rceil$;
 $C_1 \leftarrow \{c_1, c_2, \dots, c_k\}$;
 $C_2 \leftarrow \{c_{k+1}, c_{k+2}, \dots, c_r\}$;
 $\Delta_1 \leftarrow \text{CORED}(C_{KB} \cup C_2, C_2, C_1)$;
 $\Delta_2 \leftarrow \text{CORED}(C_{KB} \cup \Delta_1, \Delta_1, C_2)$;
return($\Delta_1 \cup \Delta_2$);

Performance evaluation We evaluated the performance of the redundancy detection algorithm CoreDiag (see Table 5.6). We measured redundancy in the terms of the redundancy rate (see Formula 5.2).

$$\text{Redundancy rate} = \frac{\#\text{redundant constraints in FM}}{\#\text{constraints in FM}} \quad (5.2)$$

Feature Model	# Variables	#Constraints	Redundancy rate	Runtime (ms)
Car Selection	72	96	0.64	5070
SmartHome V. 2.2	61	63	0.29	1907
Xerox	172	205	0.71	3261

Table 5.6: Evaluation of CoreDiag with selected S.P.L.O.T. feature models

The outcome of this analysis was that all the investigated knowledge bases showed quite different degrees of redundancy (see Table 5.6). Note that

5.2 Anomaly Management

the CoreDiag algorithm is especially useful in situations where models are developed by one or a few engineers. In this case the degree of redundant constraints in the model is low. For scenarios with high redundancy rate, alternative algorithms have already been developed (see, e.g., [Felfernig et al. \[2011b\]](#)).

Next, we conducted a study to check when CoreDiag performs better compared to SEQUENTIAL. Therefore we tested both algorithms with different redundancy rates (see Figure 5.2). If the number of redundant constraints r in relation to the total number of constraints n is high, the CoreDiag algorithm performs better and has an approx. 40% runtime advantage.

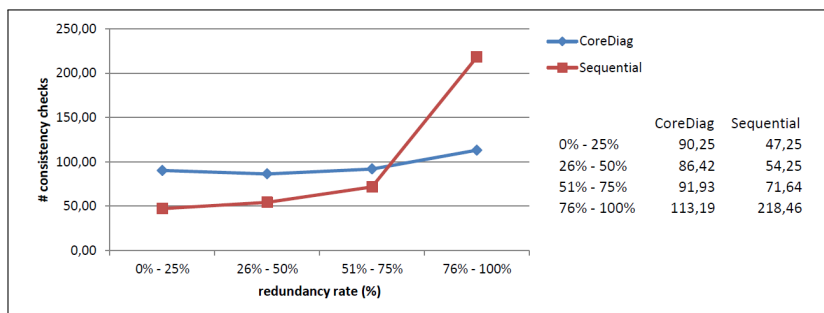


Figure 5.2: Number of consistency checks of redundancy detection algorithms

The SEQUENTIAL approach performs better if the redundancy rate is lower than 50% and loses the performance advantage if the configuration knowledge base contains between 50% and 75% redundant constraints. This confirms the complexity of CoreDiag ($2r \times \log_2(\frac{n}{r}) + 2r$) and SEQUENTIAL (n) where r is the number of redundant constraints and n is the number of all constraints in the knowledge base. Beginning with a redundancy rate $\frac{r}{n} > 0.6$, the CoreDiag algorithm has a performance advantage compared to SEQUENTIAL.

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

Each model is calculated 20 times and the configuration knowledge bases differ in the ordering of the constraints. Comparing the difference between the maximum and minimum number of consistency checks spent for the calculation of CoreDiag and SEQUENTIAL, we find out that the standard deviation between the minimal number of checks spent for the calculation of redundancies is higher when using the CoreDiag (38.67%) compared to the SEQUENTIAL algorithm (19.34%). The reason for that is the influence of the constraint order in CoreDiag and its divide-and-conquer strategy.

Assignment-based anomaly management Anomaly management research describes different approaches to detect and explain anomalies [Reinfrank et al., 2015a; Wotawa et al., 2015a]. For example, QuickXplain can detect conflicts [Junker, 2004], FastDiag finds minimal diagnoses for these conflicts [Felfernig and Schubert, 2011b], Sequential [Piette, 2008] and CoreDiag [Felfernig et al., 2011b] can remove maximal sets of constraints without changing the semantics of the knowledge base (redundancy detection). Well-formedness violations can detect domain elements which can never be selected (*dead elements*), or have to be selected (*full mandatories*), or can only exit if specific domain elements of other variables are selected as well (*unnecessary refinements*) [Reinfrank et al., 2015a].

While all of these algorithms focus on constraints, little attention has been paid to the context of assignment-based anomaly detection. Compared to constraint-based perspectives, an assignment-based view can a) find out which assignments within a constraint lead to the anomaly and b) detect more redundancies when one assignment within a constraint with more than one assignment can not be detected with common algorithms.

Alternatively, we can check the assignments within a constraint instead of the constraint itself for anomalies. Algorithm 8 gives an example for an assignment-based algorithm. This algorithm extends the *Sequential* algorithm introduced by Piette [2008]. First of all, we have to generate the negation of all constraints in the knowledge base. We denote the negation \bar{C} and define a disjunctive query of the original knowledge base $\neg c_1 \vee \neg c_2 \vee \neg c_3$.

If the negation of the knowledge base in combination with the original knowledge base is inconsistent, s.t. $C \cup \bar{C}$ is inconsistent, the knowledge base has not changed its semantics. If a constraint will be removed from the knowledge base (but not from the negation of the knowledge base) and the combination is still inconsistent, we can say that the knowledge base has kept its semantics and the removed constraint is redundant.

While the Sequential algorithm removes constraint by constraint from C , we divide the constraint into its assignments and remove assignment by assignment. Therefore, we introduce the set $A(c)$ which describes the set of assignments of constraint $c \in C$. When we remove an assignment from $A(c)$, we next have to consider the relations between the assignments. Figure 5.3 shows the graphical representation of all constraints and their assignments in our example knowledge base 3.1 in a conjunctive order. When we remove an assignment a from $A(c)$, we will further replace the upper relation. For example, the removal of the assignment $usage_scenario = office$ of constraint c_1 replaces the upper implication \rightarrow with the top node of those elements which will not be connected to the conjunctive constraint. In our case this is relation ' \wedge '.

Algorithm 8 introduces an approach to detect redundant assignments within a knowledge base. The approach is straight forward: First we have to generate the negation of \bar{C} . Then we select constraint by constraint. For each constraint we remove assignment by assignment a . Finally we check if the knowledge base with the changed constraint c is still inconsistent with \bar{C} . If it is inconsistent, we can say that the removed assignment a is redundant.

Figure 5.4 shows the redundant assignments of our example knowledge base. In the first row we see the original constraints and the result for the *usage_scenario* variable (green box). Then we remove assignment by assignment and see the result of the constraints in the column *result*. The yellow boxes suggest that the adapted constraints lead to the same semantics as the original knowledge base. We can remove $cpu_cores = 2$ from constraint

Algorithm 8 AssignmentSequential

function ASSIGNMENTSEQUENTIAL(KB): R

▷ KB : knowledge base

$\bar{C} = \neg c_1 \vee \neg c_2 \vee \neg c_3$

$R = \emptyset$

for all $c \in C$ **do**

for all $a \in A(c)$ **do**

$A.remove(a)$

if $(C \cup \bar{C})$ is inconsistent **then**

$R.add(a)$

else

$A.add(a)$

end if

end for

end for

return R

end function

c_1 and the assignments $price < 999$ and $cpu_cores = 4$ from filter constraint c_2 without changing the semantics of the knowledge base.

Similar adaptations can also be done, e.g., for QuickXPlain Junker [2004], FastDiag Felfernig et al. [2012b], and CoreDiag Felfernig et al. [2011b]. As these algorithms use a divide and conquer approach based on constraints, future research can also consider assignments instead of constraints to calculate the anomalies.

5.2.3 Well-formedness Violations

Next, we describe the algorithms to detect well-formedness violations. First, we detect *dead domain elements* (Definition 9) with Algorithm 9. The algorithm takes sets of constraints (C), variables (V), and their domains (D) as input parameters and returns a set of variable assignments. Each

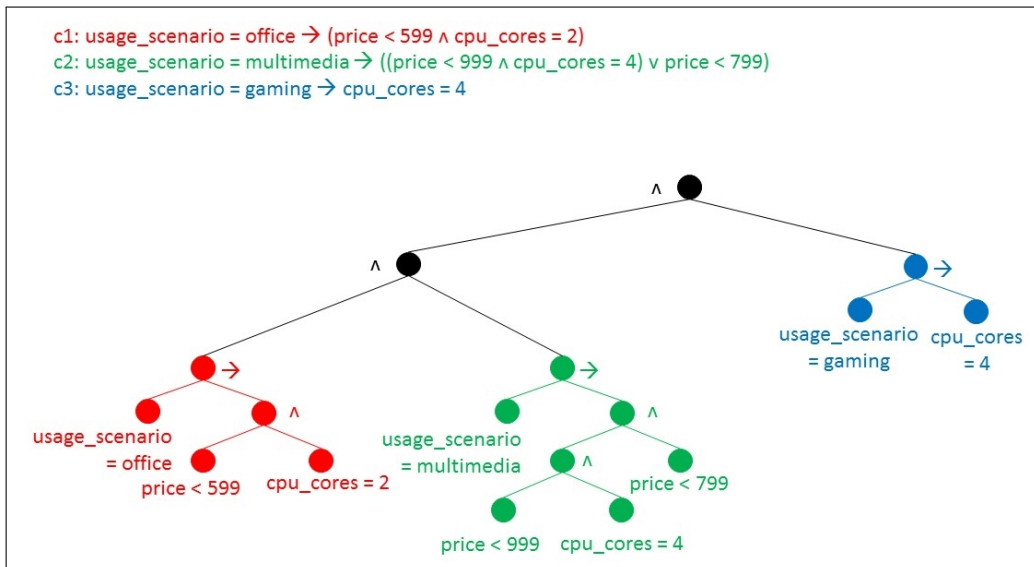


Figure 5.3: Graphical representation for detecting redundant assignments: conjunctive query of all constraints $c \in C$ in our example knowledge base. In Algorithm 8 we remove assignment by assignment from the knowledge base and check the consistency instead of the whole constraint (c_1, c_2, c_3) .

of the assignments can never be consistent with C . The suggestion for the knowledge engineer is that the domain elements, which will be returned by the algorithm, can be deleted.

While we can evaluate if a domain element can never be in a consistent instance, we can also check if a domain element must be in a consistent instance of a knowledge base. We denote such domain elements as *full mandatory* (see Definition 10). Algorithm 10 checks whether the knowledge base will be inconsistent if the domain element dom_j is not selected.

If variable v_i contains a full mandatory domain element, we can say that each other domain element of v_i is a dead element. If a domain element is full mandatory, we suggest the knowledge engineer to delete all other domain elements or to remove the variable itself.

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

c1	c2	c3	result	redundant
usage_scenario = office → (price < 599 ∧ cpu_cores = 2)	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) ∨ price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1 p1, p3	
price < 599 ∧ cpu_cores = 2	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) ∨ price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1 p1, p3	
usage_scenario = office → cpu_cores = 2	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) ∨ price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0, p2 p0, p1 p1, p3	
usage_scenario = office → price < 599	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) ∨ price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1 p1, p3	cpu_cores = 2
usage_scenario = office → price < 599	((price < 999 ∧ cpu_cores = 4) ∨ price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1 p1	
usage_scenario = office → price < 599	usage_scenario = multimedia → (cpu_cores = 4 ∨ price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1 p1, p3	price < 999
usage_scenario = office → price < 599	usage_scenario = multimedia → price < 799	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1 p1, p3	cpu_cores = 4
usage_scenario = office → price < 599	usage_scenario = multimedia	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1, p2, p3 p0 p1	
usage_scenario = office → price < 599	usage_scenario = multimedia → price < 799	cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p1 p1, p3 p0	
usage_scenario = office → price < 599	usage_scenario = multimedia → price < 799	usage_scenario = gaming	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming p0 p0, p1, p2, p3	

Figure 5.4: Detection of redundant assignments. The columns c_1 , c_2 , and c_3 show the constraints when one assignment will be removed. The first row shows the results (green background) of the original constraints. The yellow background suggests that the removal of the assignments leads to the same results.

Finally, we introduce an algorithm to detect *unnecessary refinements* between variables (see Definition 11). Algorithm 11 returns a set of constraints. Each of these constraints describe one unnecessary refinement between two variables and each domain element between both variables. The assignments between the variables are conjunctive and each domain element of variable v_i is in a disjunctive order, e.g. $(v_i = val_{i1} \wedge v_j = val_{j1}) \vee (v_i = val_{i2} \wedge v_j = val_{j2}) \vee (v_i = val_{i3} \wedge v_j = val_{j3})$.

The performance of this algorithm depends on the number of variables, their domain size, the number of unnecessary refinements, and the performance of the solver. In our study with 14 knowledge bases (up to 34 variables and domain sizes from two to 47) the detection of unnecessary refinements requires up to 375 ms (with 42 unnecessary refinements) for the detection of all possible unnecessary refinements (Intel Xeon @ 2.4Ghz, 6 cores, 24GB RAM).

Algorithm 9 DeadDomainElement (V, D, C): Δ

▷ V : knowledge base variables
 ▷ D : knowledge base domains
 ▷ C : knowledge base constraints
 ▷ Δ set with inconsistent variable assignments

```

for all  $v_i \in V$  do
  for all  $dom_j \in dom(v_i)$  do
     $C' = C \cup \{v_i = dom_j\}$ 
    if  $inconsistent(C')$  then
       $\Delta \leftarrow \{v_i = dom_j\}$ 
    end if
  end for
end for
return  $\Delta$ 
  
```

Besides the detection of dependencies between elements within a constraint-based knowledge based on collaborative and content-based recommendations (Section 5.1) and anomalies, we can also evaluate how variables (and constraints) are influencing each other. How to measure the interference of two variables or constraints will be described in the next section.

5.3 Dependency Detection⁴

To increase the understandability of a knowledge base for knowledge engineers we can evaluate the influence between variables. For example, if a knowledge engineer adds a new domain element '8' to variable cpu_{cores} she is maybe also interested in relationships to the variable RAM .

In this section we present Gibbs' sampling to measure the intensity of relationships between variables. Based on randomly generated constraints we use consistency checks (see Definition 3) to determine the relationship

⁴This Section is based on Reinfrank et al. [2015a].

Algorithm 10 FullMandatory (V, D, C): Δ

▷ V : knowledge base variables
 ▷ D : knowledge base domains
 ▷ C : knowledge base constraints
 ▷ Δ set with inconsistent variable assignments

```

for all  $v_i \in V$  do
  for all  $dom_j \in dom(v_i)$  do
     $C' = C \cup \{v_i \neq dom_j\}$ 
    if  $inconsistent(C')$  then
       $\Delta \leftarrow \{v_i \neq dom_j\}$ 
    end if
  end for
end for
return  $\Delta$ 
  
```

between variables. We measure the relationship of v_a and v_b based on the number of consistent checks compared to the overall number of consistency checks and each consistency check contains an assignment for variable v_a and v_b . While doing all possible consistency checks for constraint-based product configuration systems is possible in an acceptable time period, constraint-based configuration systems needs an unacceptable time period to evaluate all possible consistency checks. How Gibbs's sampling can be used for dependency detection in product configuration knowledge bases will be described in the following.

The general purpose is the following: when we want to determine the relationship between two variables v_a and v_b , we generate assignments for all possible combinations of their domain elements and add one of these assignments to C_{KB} . An assignment is a constraint which contains one variable v , one domain element $d \in dom(v)$, and a relationship between variable and domain element v_r (see Section 3.1). Examples for assignments are $eBike = true$ and $BikeType = MountainBike$. Additionally, we generate further randomly generated assignments for other variables in the knowledge base and check, if the knowledge base is consistent. When we calculate $consistentchecks / (consistentchecks + inconsistentchecks)$, we can say,

Algorithm 11 UnnecessaryRefinement (C, V): Δ

▷ V : knowledge base variables
 ▷ D : knowledge base domains
 ▷ C : knowledge base constraints
 ▷ Δ set with constraints

```

for all  $v_i \in V$  do
  for all  $v_j \in V | v_i \neq v_j$  do
     $A = \emptyset$ ;
    ▷ set with assignments
    for all  $dom_k \in dom(v_i)$  do
       $dompair = false$ ;
       $C' \leftarrow C \cup \{v_i = dom_k\}$ 
      for all  $dom_l \in dom(v_j)$  do
         $C'' \leftarrow C' \cup \{v_j \neq dom_l\}$ 
        if  $inconsistent(C'') \wedge dompair = false$  then
           $dompair = true$ ;
           $A \leftarrow A \cup \{v_i = dom_k \wedge v_j = dom_l\}$ 
        end if
      end for
    end for
    if  $|A| = |dom(v_i)|$  then
       $\Delta \leftarrow \Delta \cup disjunctive(A)$ 
    end if
  end for
end for
return  $\Delta$ 

```

that values near to 0 means a weak relationship, because whenever v_a and v_b have a specific assignment, the knowledge base becomes inconsistent.⁵ A value nearer to 1⁶ means also a weak dependency between both domain elements because the assignments of the domain elements of both variables seems not to have any influence on the consistency of the knowledge base.

⁵Note, that a check for dead domain elements should be done before doing a dependency detection.

⁶Note, that the value 1 is only possible, if no constraint in the knowledge base influences the variables. In practice, we can use the coverage metric (see Section 5.4.3) as the maximum for the relationship between two variables.

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

If the value is in the middle of the range, we can say, that there is a high influence of the combination of these two assignments and we have to consider this relationship when we have to maintain one of this domain elements in the knowledge base.⁷

Algorithm 12 describes Gibbs' sampling, shows the basic algorithm for estimating the consistency rate for a set of assignments, and is divided into three functions.

Algorithm 12 GibbsSampling

```
1: function GIBBS(KB, A):  $\Delta$ 
2:   CC =  $\emptyset$  ▷ set of consistency check results {0, 1}
3:   mincalls = 200 ▷ constant
4:   threshold = 0.01 ▷ constant
5:   consistent = 0
6:   verify = Double.Max_Value
7:   while  $n < \text{mincalls} \vee \text{verify} > \text{threshold}$  do
8:     randA =  $A \cup \text{GENERATERANDASSIGN}(KB)$ 
9:     C.addAll(randA) ▷  $C \in KB$ 
10:    if isConsistent(KB) then
11:      consistent ++
12:      CC.add(1)
13:    else
14:      CC.add(0)
15:    end if
16:    C.removeAll(randA)
17:    verify = VERIFYCHECKS(CC)
18:    n ++
19:  end while
20:  return consistent / n
21: end function
```

⁷An example for the graphical representation of the results is given in Figure 6.11.

5.3 Dependency Detection

The function $Gibbs(KB, A)$ is the main function of this algorithm. It has a knowledge base KB and a set of assignments A as input. The knowledge base contains sets of variables $V \in KB$ and constraints $C \in KB$. The set CC (checks) contains all results from consistency checks. A consistency check is either consistent (1) or inconsistent (0). The number of minimum calls is constant and given in variable $mincalls$. The total number of consistent checks is given in variable $consistent$. $threshold$ is a constant and required for testing if the current set of consistency checks has a high accuracy. If the variable $verify$ is greater than the $threshold$ we can not guarantee, that the current result is accurate. Therefore, we have to execute the loop again. In the while-loop we first have to generate a set of new random assignments. Since assignments are also special types of constraints, we add them to the set $C \in KB$ and do a consistency check again. If $randA \cup C \in KB$ is consistent, we add 1 to the set CC and increment the variable $consistent$. Otherwise we add 0 to the set CC . Finally, we verify all previous consistency checks. If the variable $verify$ is lower than the variable $threshold$ and we have more consistency checks than $mincalls$, we can return the number of consistent checks divided by the total number of checks.

```
22: function GENERATERANDASSIGN( $KB$ ): $A$ 
23:    $A = \emptyset$  ▷  $A$ : set of assignments
24:    $n = 0 < Random(C) \leq |C|$ : ▷ generate n assignments
25:   for  $i = 0; i < n; i++$  do
26:      $a_v = Random(V)$  ▷  $V \in KB$ 
27:      $a_r = Random(R)$ 
28:      $a_d = Random(dom(a_v))$ 
29:      $A.add(a)$ 
30:   end for
31:   return  $A$ 
32: end function
```

The function $generateRandAssign(KB)$ is responsible for the generation of new assignments. $Random(C)$ returns the number of assignments which has to be generated randomly. $Random(V)$ takes a variable from the knowledge base. If the variable is already part of another assignment, the variable will not be used again. $Random(R)$ selects a relation between the variable

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

and the domain elements. In our case, variables can have textual domain elements (e.g. the brand of a bike) or numeric domain elements (e.g. the price of a bike). While the set of relations for textual domain elements is $R = \{=, \neq\}$, the set is extended to $R = \{=, \neq, <, \leq, >, \geq\}$ for numerical domain elements. Finally, $Random(dom(a_v))$ selects a domain element from $dom(a_v)$ randomly (see Section 3.1).

```
33: function VERIFYCHECKS(CC): $\Delta$ 
34:    $CC_1 = CC.split(0, |CC|/2)$ 
35:    $CC_2 = CC.split(|CC|/2 + 1, |CC|)$ 
36:    $mean1 = mean(CC_1)$ 
37:    $mean2 = mean(CC_2)$ 
38:   if  $mean1 \geq mean2$  then
39:     return  $mean1 - mean2$ 
40:   else
41:     return  $mean2 - mean1$ 
42:   end if
43: end function
```

Function *verifyChecks*(CC) tests if the number of consistent and inconsistent checks is normally distributed. Therefore, we first divide the set with the consistency check results CC into two parts. We evaluate the mean of both sets CC_1 and CC_2 and test if both mean values are near to each other. If they have nearly the same value, we can say that the consistent checks are normally distributed in both sets and return the difference between *mean1* and *mean2*.

In this section we presented a usage scenario for Gibbs' sampling for detecting the dependencies between variables. Basically, we can say, that Gibbs' sampling can be used whenever an exact calculation is not necessary and calculating a metric is too expensive (e.g., consistency checks for all variants in constraint-based product configuration systems). For example, with Gibbs' sampling we can ...

5.4 A Goal-Question-Metrics Model for Product Configuration Knowledge Bases

- ... generate test cases for boundary value analysis [Reinfrank et al., 2015c].
- ... rank diagnoses and conflicts (assuming that a knowledge base with nearly the same coverage compared to the current knowledge base is preferred).
- ... generate reports for variety management (e.g. 'How many bikes can be used for *Competition*, *EverydayLife*, and *HillClimbing?*').
- ... estimate the coverage (number of consistent instances compared to all instances) and evaluate the knowledge base which will be described in the next section.

5.4 A Goal-Question-Metrics Model for Product Configuration Knowledge Bases⁸

For the overview of the metrics for configuration knowledge bases we use the GQM method. For each goal we use a set of questions to define the achievement of each goal. It is also necessary that the goals, questions, and metrics can be calculated automatically and with explanations [Bagheri and Gasevic, 2011; Salvetto et al., 2004].

In this section we first give an overview of the possible goals for configuration knowledge bases. Thereafter we give an overview of the questions concerning (configuration) knowledge bases. Finally we operationalize the questions by listing metrics for configuration knowledge bases.

5.4.1 Goals for Product Configuration Knowledge Bases

Nabil et al. [2008] define five basic goals for knowledge bases. *Reusability* means that the knowledge base can be reused in another application area. The *flexibility* defines the possibility to change the semantics of the configuration knowledge base. *Understandability* defines the possibility that knowledge engineers have correct assumptions. *Functionality* describes the

⁸This Section is based on Reinfrank et al. [2015b].

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

applicability of the knowledge base. For example, if the model does not describe the real product assortment, the knowledge base has no functionality. *Extendability* describes the possibility to extend the knowledge base. In our example (see Section 3.3) we can extend the model by adding the bike type ($V' = V \cup \{Gears\}; D' = D \cup \{dom(Gears) = \{1, 6, 18, 21, 24, 27\}\}$);).

Lethbridge [1998] identifies three goals for knowledge bases. First, it is necessary that knowledge engineers can *monitor* their work. Therefore it is necessary to offer baselines for its continuous improvement. Another aspect is the support for knowledge engineers when they *maintain* a knowledge base. Finally, Lethbridge also focuses on the *understandability* of knowledge bases.

From the perspective of software product lines there are three goals: *Analyzability* focuses on the capability of a system to be diagnosed for anomalies. *Changeability* is the possibility and ease of change in a model when modifications are necessary. *Understandability* also means the likelihood that knowledge engineers and designers understand the knowledge base [Bagheri and Gasevic, 2011].

To sum up, we define the following goals for product configuration knowledge bases:

- A configuration knowledge base must be **maintainable**, such that it is easy to change the semantics of the knowledge base in a desired manner [Bagheri and Gasevic, 2011; Nabil et al., 2008].
- A configuration knowledge base must be **understandable**, such that the effort for a maintainability task for a knowledge engineer can be evaluated [Bagheri and Gasevic, 2011; Nabil et al., 2008; Lethbridge, 1998].
- A configuration knowledge base must be **functional**, such that it represents a part of the real world (e.g. a bike configuration knowledge base) [Nabil et al., 2008].

5.4 A Goal-Question-Metrics Model for Product Configuration Knowledge Bases

5.4.2 Questions for Product Configuration Knowledge Bases

Having defined the goals for configuration knowledge bases, we now describe the questions relating to one or more goals. The concordance with the application will be defined by the *completeness*. It suggests the applicability of the current state of the knowledge base for representing the application area. For instance, in our example 3.2 a *Bike* can be used for either *Competition* or *EverydayLife*. If it is not possible to combine those *Usages*, the coverage is high. If a bike can be used for both usage scenarios, the model does not represent the application area and the coverage will be low.

Q1: Is the configuration knowledge base complete?

Anomalies are a well researched area in the context of configuration knowledge bases [Preece, 1998]. The term *anomalies* is used synonymously for errors and subsumes the terms inconsistencies, redundancies, and well-formedness violations. Errors can have an impact on each of the goals, since they have negative impacts on reusability, maintainability, and understandability. They can also have a negative impact on the functionality, if there exists an inconsistency in the knowledge base.

Q2: Does the configuration knowledge base contain anomalies?

The *performance* describes the time which is required to calculate characteristics of a knowledge base. These characteristics are, e.g., error checking, calculating consistent configurations, and generating user recommendations. This performance mainly influences the functionality of a system (latency) and the reusability.

Q3: Does the configuration knowledge base have an admissible performance?

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

If it is necessary to develop and maintain the knowledge base, a high *modifiability* will help to reduce the effort for the update operation. The modifiability has a positive impact on the reusability and maintainability of a knowledge base. For example, when updating redundant constraints in a knowledge base (e.g. constraint c_7 in the example 3.2, it is probably necessary to update the redundant constraints (c_2), too. This may lead to a low functionality because the knowledge base does not have the correct behavior.

Q4: Is the configuration knowledge base modifiable?

The development effort describes the effort when updating a configuration knowledge base. This effort contains the time for the update operation. It includes the update of the semantics of the knowledge base and the time, which is required to remove all new errors. This effort has an impact on the maintainability and reusability of a knowledge base and is mainly influenced by the *understandability* of a configuration knowledge base.

Q5: Is the configuration knowledge base understandable?

Not every goal has a relationship with every question. In Table 5.7 we give an overview of the relationship between goals and questions.

Question / Goal	MT	US	FT
Q1 (completeness)			+
Q2 (anomalies)	-	-	-
Q3 (performance)			+
Q4 (modifiability)	+		
Q5 (understandability)		+	

Table 5.7: Relations between goals and metrics (MT = maintainability, US = usability, FT = functionality)

5.4.3 Metrics for Product Configuration Knowledge Bases

The metrics are based on a literature review focusing on knowledge engineering [Felfernig et al., 2012a; Nabil et al., 2008; Baumeister et al., 2004; Mehrotra et al., 2002; Blythe et al., 2001a; Lethbridge, 1998; Preece, 1998; Barr, 1997; Preece et al., 1997; Preece and Shinghal, 1994; van Melle et al., 1984] as well as on software product line engineering [Bagheri and Gasevic, 2011; Benavides et al., 2010; Hartmann and Trew, 2008; Lauenroth and Pohl, 2007]. The assumptions in this section are based on the literature of configuration knowledge bases and other research areas like feature models and software product lines, software engineering, and rule-based knowledge bases.

Having defined the questions for configuration knowledge bases, the next task is to quantify the metrics. Therefore, we describe possible metrics for configuration knowledge bases. Most of the metrics require a consistent product configuration knowledge base. Therefore, we use example 3.2. Example 3.3 is used to describe metrics for inconsistencies. The metrics are based on literature study in configuration, feature model, and software engineering research areas.

The next list shows some metrics derived from MOOSE and function point analysis [Bagheri and Gasevic, 2011; Salvetto et al., 2004; Lethbridge, 1998; Chidamber and Kemerer, 1994]:

- **Number of variables** $|V|$: $|V| = 6$.
- **Average domain size**: $domsize = \frac{\sum_{v_i \in V} |dom(v_i)|}{|V|} = 2.6\dot{6}$
- **Number of constraints**: $|C| = 9$

The **number of minimal conflicts** $|CS|$ is the first anomaly metric. In our example 3.3 (see Section 3.3) we have two minimal conflict sets (see Section 4.1), such that $|CS| = 2$. We can also evaluate the smallest number of constraints in a conflict set. The lowest number of constraints in a conflict set

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

CS is the **minimal cardinality of conflict sets** $MCCS$ and can be defined as $\#CS_j : |CS_j| < |CS_i|$. The example has a minimal cardinality $MCCS = 1$.

We can also evaluate diagnoses for knowledge bases. For example, with the FastDiag algorithm [Felfernig et al., 2012b; Felfernig and Schubert, 2011b] we can calculate the **number of diagnoses** $|\Delta|$ and the number of constraints in a **minimal cardinality diagnosis** MCD . A minimal cardinality diagnosis Δ_i is a minimal diagnosis which has the property of having the smallest number of constraints in the diagnosis, such that, $\# \Delta_j : |\Delta_j| < |\Delta_i|$. Example 3.3 described in Section 3.3, contains 2 minimal diagnoses ($\Delta_1 = \{c_9, c_{10}\}, \Delta_2 = \{c_{10}, c_{11}\}, |\Delta| = 2$) and a minimal cardinality diagnosis of ($MCD = 2$).

The number of redundant constraints can also be used as a measure for knowledge bases [Felfernig et al., 2011b; Piette, 2008; Preece, 1998; Preece and Shinghal, 1994] if the configuration knowledge base is consistent.⁹ The number of sets of **redundant constraints** is denoted as $|R|$ and the **maximum cardinality of a redundancy set** $R_i = 2$. We calculate the maximum cardinality for R_i by checking if there exists another set R_j which has a bigger cardinality, such that R_i has the property of having the maximum cardinality, iff $\#R_j |R_j| < |R_i|$. Example 3.2 in Section 3.3 contains one set with redundant constraints ($R_1 = \{c_3, c_7\}, |R| = 1$) and the maximum cardinality of these sets is 2 ($MCR = |2|$).

A domain element $dom_i \in dom(v_j)$ is a **dead domain element** iff there does not exist a valid configuration with this assignment, such that $C \cup \{v_j = dom_i\}$ is inconsistent [Benavides et al., 2010; Baumeister et al., 2004]. We use the sum of all dead elements as a metric $0 < DE < 1$ by using Equation 5.3 where a value nearer 0 means that there are no or less dead elements and a value nearer to 1 means that a high number of domain elements in the knowledge base can not be selected in a consistent configuration knowledge base.

⁹Note that we are now using Example 1 for the next metrics (see Section 3.3).

5.4 A Goal-Question-Metrics Model for Product Configuration Knowledge Bases

$$DE = \frac{\sum_{v_i \in V} \sum_{d_j \in \text{dom}(v_i)} \begin{cases} 0 & C \cup \{v_i = d_j\} \neq \emptyset \\ 1 & \text{else} \end{cases}}{|V| \times \text{domsize}} \quad (5.3)$$

On the other hand, a domain element can be **full mandatory** (*FM*). Full mandatory means that there does not exist a consistent instance of the knowledge base where this domain element is not selected, formally described in Equation 5.4.

$$FM = \frac{\sum_{v_i \in V} \sum_{d_j \in \text{dom}(v_i)} \begin{cases} 0 & C \cup \{v_i \neq d_j\} = \emptyset \\ 1 & \text{else} \end{cases}}{|V| \times \text{domsize}} \quad (5.4)$$

In feature models [Benavides et al., 2010] each domain has exactly two values $\{true, false\}$. For domains with two domain elements we can say that whenever a domain element is dead, the other domain element becomes full mandatory automatically. When domains have more than two values, it can be the case that a domain element is dead but there is no other domain value with the property of being a full mandatory domain element.

The third well-formedness violation is called **unnecessary refinement** (*UR*). Such a violation occurs when there are two variables and the domain element of the first variable in a valid configuration can be suggested by the assignment of a second variable. An unnecessary refinement can be described as $\text{dom}(v_i) \rightarrow \text{dom}(v_j)$.

In example 3.2 we can say that the variables *BikeType* and *Usage* are unnecessary refined because whenever $\text{Usage} = \text{Competition}$ then $\text{BikeType} = \text{RacerBike}$, $\text{Usage} = \text{EverydayLife}$ then $\text{BikeType} = \text{CityBike}$, and $\text{Usage} =$

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

HillClimbing then $BikeType = MountainBike$. In that case we can recommend that the variable $BikeType$ can be removed and its occurrences in constraints can be replaced by the variable $Usage$. If both variables are required (e.g., because the customer should be asked for $Usage$ and manufacturing is requiring the term $BikeType$) we recommend to remove one of these variables from the knowledge base temporarily for improving the performance of solvers.

The **restriction rate** RR compares the number of constraints with the number of variables. In example 3.2 the restriction rate $RR = \frac{|C|}{|V|} = \frac{9}{8} = 1.125$. A value greater than 1 means that there is a high restriction [Bagheri and Gasevic, 2011; Lethbridge, 1998].

The metric RR is influenced by the design of the knowledge base. For example, while one knowledge engineer requires a single constraint for subsuming the constraints $c_0 \wedge c_1 \wedge c_2 \wedge c_3$, another knowledge engineer is using four single constraints. To consider these different design approaches in the metric, the **restriction rate** RR_2 is considering the number of variables in a constraint, such that, $RR_2 = \frac{\sum_{c_i \in C} \frac{\#vars(c_i)}{\#vars(C)} |C|}{|V|}$ where $\#vars(c_i)$ is the number of variables in c_i .

Another metric from the domain of software engineering is the **variable inheritance factor** VIF [Abreu and Melo, 1996]. Adapted for configuration knowledge bases, we define VIF as the number of constraints in which a variable v_i appears related to the number of constraints, e.g., $VIF(eBike) = 0.22$ because the variable $eBike$ appears in two constraints and $|C| = 9$ (see Equation 5.5).

$$VIF(v_i) = \frac{\sum_{c_i \in C} \begin{cases} 1 & v_i \in c_i \\ 0 & \text{else} \end{cases}}{|C|} \quad (5.5)$$

5.4 A Goal-Question-Metrics Model for Product Configuration Knowledge Bases

To receive a knowledge base metric we calculate the VIF_{all} for all variables. Calculating the arithmetic mean of the VIF_{all} of all variables, we can evaluate the importance distribution of all variables (see Equation 5.6). A value near to 0 means that all variables have the same importance and should be considered in the same way. On the other hand, a high value means that there are some important and less important variables in the knowledge base. In such cases it makes sense to focus on the important variables when maintaining the knowledge base.

$$VIF_{all} = \sum_{v_i \in V} \frac{\sqrt{(VIF(v_i) - \frac{\sum_{v_j \in V} VIF(v_j)}{|V|})^2}}{|V|} \quad (5.6)$$

Finally, we evaluate the metric **coverage**. The *coverage* measures the number of all consistent complete configurations (see Section 3.2) compared to the maximum number of complete configurations in a knowledge base. In our example 3.2 the maximum number of configurations is $\prod_{i=0}^{|V|} |dom(v_i)| = 324$ (6 variables multiplied with their domain size). This will be compared with the number of consistent configurations. In our example we have the following seven consistent configurations:

```

UniCycle = false ∧ (
  Usage = Competition ∧ BikeType = RacerBike ∧ TireWidth = 57mm ∧
  eBike = false ∧ FrameSize = 60cm
) ∨ (
  Usage = EverydayLife ∧ BikeType = CityBike ∧ TireWidth = 37mm ∧ (
    (eBike = false ∧ FrameSize = 50cm) ∨
    (eBike = false ∧ FrameSize = 60cm) ∨
    (eBike = true ∧ FrameSize = 50cm) ∨
    (eBike = true ∧ FrameSize = 50cm)
  )
) ∨ (
  Usage = HillClimbing ∧ BikeType = MountainBike ∧ TireWidth = 57mm ∧
  eBike = false ∧ (FrameSize = 50cm ∨ FrameSize = 60cm)
)

```

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

Now we can compare the number of consistent configurations (= 7) with the number of all configurations (= 324). This leads to a coverage of $7/324 * 100 \approx 2.16\%$ which is very low and the example configuration knowledge base is very restrictive. For the example knowledge base it is quite easy to evaluate all possible combinations of variables and domain elements. For knowledge bases with more variables, domain elements, and constraints we have millions and more possible combinations of variable assignments. For such scenarios we introduced the **simulation technique** in the context of knowledge based systems to approximate the *coverage*. For a detailed description to approximate this metric in large configuration knowledge bases we refer the reader to [Reinfrank et al. \[2015a\]](#) and to the previous Section 5.3.

Finally, we can refer the questions to the metrics. Table 5.8 gives an overview of the relationships between the questions and the metrics.

For a detailed description of the calculation of metrics focusing on anomalies (conflicts, redundancies, and well-formedness violations) and the *coverage* metric we refer the reader to [[Reinfrank et al., 2015a](#)].

5.4.4 Discussion

In this Section we want to discuss relevant aspects of several metrics and give an insight into the implementation of the goal-question-metrics in the iCone interface (see Section 6.2).

Most of the research in the area of configuration knowledge engineering focuses on the area of verifying configuration knowledge bases (goal **functionality**, see Section 4) and ignores the question how to validate the knowledge base [[Preece, 1998](#)] (**maintainability** and **understandability**). [Felfernig et al. \[2010\]](#) present an empirical study about the understandability

5.4 A Goal-Question-Metrics Model for Product Configuration Knowledge Bases

	Q1	Q2	Q3	Q4	Q5
$ V $	+		-		
<i>domsize</i>	+		-		
$ C $	+		-		
$ CS $	-	-		-	
$ \Delta $	-	-		-	
<i>MCCS</i>					+
<i>MCA</i>					+
$ R $		-	-	-	
<i>MCR</i>					+
<i>DE</i>		-	-	-	-
<i>FM</i>		-	-	-	-
<i>UR</i>		-	-	-	-
<i>RR</i>				-	-
<i>RR₂</i>				-	-
<i>VIF_{all}</i>				-	-
<i>Coverage</i>				-	-

Table 5.8: Relations between metrics (rows) and questions (columns). A '-' means that the metric has a negative impact on the question, '+' represents a positive impact.

of constraints in knowledge bases but there does not exist a metric for the understandability of constraints and the knowledge base.

Briand et al. [1999] measured the effects of the structural complexity of software and its relationship to the **maintainability** of software. Bagheri and Gasevic [2011] transferred this model into the area of feature models and found out that the number of leaf features, the cyclomatic complexity, the flexibility of configuration, and the number of valid configurations influence the maintainability of feature models. While the simple metrics are easy to transfer into configuration knowledge bases, the depth of a tree or the number of valid configurations can not be calculated.

The number of **redundant constraints** is an important metric since a low number of redundant constraints can improve the maintenance task,

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

simplify the understandability, and reduce the time for calculating valid configurations. An important issue in that case is that redundant constraints can also improve the understandability of a configuration knowledge base. If a redundant constraint is declared as a desired redundant constraint, the metric should not contain such constraints but should list it as a desired redundancy.

In a simple configuration knowledge base like example 3.2 it is easy to calculate the consistency of each possible configuration for the *coverage* metric. For example, in a configuration knowledge base with a medium number of variables (e.g. 10) and average domain size (e.g. 5) we have approximately 10M possible configurations. Since it is not possible to calculate so many possible configurations in real-time, we developed a simulation strategy to approximate the number of consistent configurations. For a detailed description of the simulation strategy we refer the reader to [Reinfrank et al., 2015c] and Section 5.3.

Showing the GQM to knowledge engineers can help to understand and maintain the configuration knowledge base, and it is also important for interpreting the results. Therefore we implemented a history for each metric in our iCone-interface (see Section 6.2). When updates in a configuration knowledge base in the iCone-system are saved, a new version of the knowledge base will be created and metrics will be actualized. In Figure 5.5 we can see the changes of the value of the *DEAD* elements metric.

Felfernig [2004] gives an overview of the usage of **function point analysis** for configuration knowledge bases. Therefore he analyzed the input interface of a configuration knowledge base for customers and the complexity of a configuration knowledge base. They use the customer requirements as external input (EI), the data required by the user as external query (EQ), the consistent domain elements of variables as external output (EO), knowledge elements as internal logical files (ILF), and external information like the product assortment from an ERP-system as external interface file (EIF). While this approach takes input and output into account, it does not evaluate

5.4 A Goal-Question-Metrics Model for Product Configuration Knowledge Bases

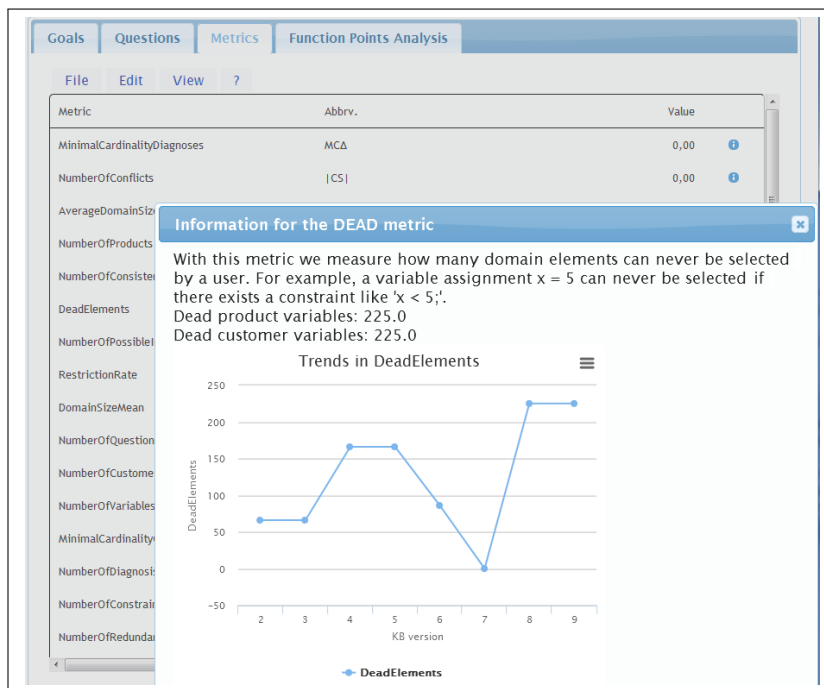


Figure 5.5: Visualization of changes for the metric *DEAD*. The y-axis shows the number of *DEAD* variables in each version of the configuration knowledge base (x-axis).

the quality (e.g., the number of *dead* domain elements) of the input and output.

We have implemented the GQM and the FPA approach in our iCone implementation [Wotawa et al., 2015a] (see Section 6.2). Table 5.9 gives an overview of the performance. Note, that the time contains the calculation / approximation of the metrics and the calculation of all anomalies. The notebook domain is calculated six times and the mobile phone domain is calculated seven times.

In this Section we gave an overview of metrics in configuration knowledge bases, focusing on **knowledge base engineering processes** [Studer et al., 1998]. We can measure the metrics of an existing configuration knowledge base but we can not identify the causes of bad configuration knowledge

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

	Notebooks	Mobile phones
Product variants	115.00	13,999.00
Product variables	28.00	34.00
product variable domain sizes	1.00 - 45.00	2.00 - 47.00
Customer variables	4.00	5.00
avg. customer variable dom. size	3.75	4.00
constraints	12.00	8.00
min. calc. time	669 msec.	6,811 msec.
max. calc. time	1,715 msec.	18,643 msec.
median calc. time	1,213 msec.	10,842 msec.
mean calc. time	1,252 msec.	11,307 msec.

Table 5.9: Duration for the calculation of all anomalies (conflicts, diagnoses, redundancies, well-formedness violations), metrics, goal-question-metrics and function-point-analysis for two configuration knowledge bases (notebooks and mobile phones)

base engineering. To give recommendations for optimizing the knowledge base engineering process, we have to observe the whole process (see Section 5.6).

This thesis describes several techniques to support knowledge engineers when they have to maintain a part of the knowledge base. On the other hand, we can do repair actions automatically (Section 4.3). In the next section, we present techniques to generate test cases automatically.

5.5 Test Case Generation¹⁰

In this Section we want to describe a basic approach to generate test cases for constraint-based product configuration systems based on automated test case generation and collaborative verification.

¹⁰This Section is based on Reinfrank et al. [2015c].

5.5 Test Case Generation

In software engineering, boundary value analysis are *those situations directly on, above, and beneath the edges of input equivalence classes* [Myers et al., 2012]. Using this type of software testing in the context of configuration systems, we can say that the *edges* are within variable assignments. In our example 3.1, if $price = 399$ is consistent, $price = 599$ is consistent too, and $price = 799$ is inconsistent, the boundary would be between the domain elements 599 and 799. In Figure 6.11 we can see that, under circumstances, some combinations are inconsistent (e.g. $usage_scenario = gaming \wedge cpu_cores = 2$) and some are consistent (e.g. $usage_scenario = multimedia \wedge cpu_cores = 2$). We can use the simulation technology (see Section 5.3) to generate various sets of filter constraints to get some boundaries. Table 5.10 shows a list of randomly generated test cases. Note that the number of assignments in the test case can be different (see Algorithm 12).

<i>tc</i>	<i>filterconstraint</i>	<i>coverage</i>
t_0	$cpu_cores = 2 \wedge$ $usage_scenario = office$	0.50
t_1	$cpu_cores = 2 \wedge$ $usage_scenario = multimedia$	0.50
t_2	$price = 799 \wedge$ $usage_scenario = gaming$	0.00
t_3	$price = 599 \wedge$ $usage_scenario = gaming$	0.50
t_4	$cpu_cores = 4 \wedge$ $usage_scenario = multimedia$	0.50
t_5	$cpu_cores = 4$	~ 0.54

Table 5.10: An example for randomly generated test cases.

The next step is to evaluate these randomly generated boundary test cases according to the domain experts' knowledge. Our example test cases show that between the test cases t_2 and t_3 is a boundary because the coverage is different.

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

After the randomly detected boundaries via simulation we have to evaluate the boundary. Such evaluations have to be done by stakeholders of the knowledge base and can be done via micro tasks [Felfernig et al., 2014b]. In this context several stakeholders can be asked if the results of randomly generated test cases are valid or not. Such answers can be collected within a case base. Table 5.11 gives an example case base.

<i>stakeholder</i>	<i>testcase</i>	<i>correct?</i>
s_0	t_2	<i>yes</i>
s_1	t_2	<i>yes</i>
s_2	t_2	<i>yes</i>
s_3	t_2	<i>yes</i>
s_4	t_2	<i>yes</i>
s_5	t_2	<i>no</i>
s_6	t_2	<i>yes</i>
s_7	t_2	<i>yes</i>
s_0	t_3	<i>no</i>
s_1	t_3	<i>no</i>
s_2	t_3	<i>yes</i>
s_3	t_3	<i>yes</i>
s_4	t_3	<i>no</i>
s_5	t_3	<i>no</i>
s_6	t_3	<i>no</i>
s_7	t_3	<i>yes</i>

Table 5.11: An example case base for evaluating randomly generated test cases.

87.5% of the stakeholders agree that t_2 is correct which means that the test case should be inconsistent and the test case currently leads to an inconsistency. On the other hand, 62.5% of the stakeholders think that t_3 should not be consistent. This example represents a conflict between the knowledge engineers' opinions of the knowledge base. For such scenarios we have to offer relevant information to the stakeholders such as mails, forum, and content-based recommendation [Jannach et al., 2010].

5.6 Constraint-based Product Configuration system Development

Finally, a result of the discussion leads to a consistent knowledge base (constraints $c \in C$ have to be updated or removed) which represents the real product domain. The maintenance of a configuration knowledge base can be supported by explanation techniques. A detailed description of explanations for constraint-based configuration systems is given in Section 4.2.

Now we have described, how we can support knowledge engineers in their maintenance tasks. The development of such systems is out of scope of this thesis but is as important as the maintenance is. Next, we present a discussion of possible techniques to develop constraint-based product configuration systems effectively.

5.6 Constraint-based product configuration system development¹¹

A lot of research has been done in the maintenance of constraint-based systems. For example, we can evaluate the quality of knowledge bases (see Section 5.4 and Reinfrank et al. [2015b]) and check if the knowledge base has anomalies (see Section 5.1 and Reinfrank et al. [2015a]; Felfernig et al. [2013a]). Therefore, we can evaluate if we are doing the knowledge base maintenance efficiently.

Less work has been done in the context of constraint-based product configuration knowledge base development, a task which is crucial for an effective constraint-based configuration system. Next, we want to summarize previous work in the context of knowledge base development processes and try to give hints for transferring research results from the software engineering discipline into the knowledge base development research area.

¹¹This Section is based on Reinfrank et al. [2015c].

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

Development processes for constraint-based configuration systems In this Section we present an overview of current development processes for knowledge-based systems. A detailed discussion about these engineering processes is given in [Friedrich et al., 2014; Studer et al., 1998].

Common-KADS focuses on different models (organization, task, agent, communication, and expertise) of the knowledge base. For example, the expertise model tries to describe knowledge from a static, functional, and a dynamic view. While this system tries to consider all stakeholders, it does not prioritize the knowledge and does not try to solve conflicts in the knowledge before it will be transferred into a constraint-based configuration system [Schreiber et al., 1994].

The **MIKE** engineering process can be seen as an iterative process and is divided into the activities *elicitation, interpretation, formalization / operationalization, design, and implementation*. The entire development process, i.e. the sequence of knowledge acquisition, design, and implementation, is performed in a cycle inspired by a spiral model as process model. Every cycle produces a prototype as output which can be evaluated by tests in the real target environment. The evaluation of each activity will be done by domain experts. While the result of the implementation activity can be evaluated by domain experts, a deep understanding of modelling techniques is required to evaluate the results of elicitation, interpretation, and formalization activities [Angele et al., 1998].

Protege-II is used to model method and domain ontologies. A method ontology defines the concepts and relationships that are used by a problem solving method for providing its functionality. Domain ontologies define a shared conceptualization of a domain. Both ontologies can be reused in other domains which may reduce the effort to build-up a new knowledge base with similar elements [Musen et al., 1994].

5.6 Constraint-based Product Configuration system Development

Development in the Software Engineering Discipline Compared to development processes for constraint-based configuration systems, we give an overview of current trends in the engineering of such systems and create a link to the currently existing development processes for constraint-based configuration systems.

A relevant task in software engineering is **requirements engineering**. Transferring this aspect into the context of developing constraint-based configuration systems, we can say that products, product variables, questions to customers, variable domains, and filters can be functional requirements whereas interface development (e.g. to an ERP-system), performance, and collaborative development are non-functional requirements. When knowledge base engineering processes have to be finalized with a given budget and time, we also have to prioritize such requirements. Therefore, we have to rank the requirements based on their necessity and effort (time and budget) for a functional knowledge base. The prioritization should be done by different stakeholders to include as much knowledge as possible into the prioritization process.

Many different constraint-based configuration systems will be developed but each of them is developed from scratch. Similar to requirements engineering, most of the aspects of a new knowledge base are new and reuse is not possible. On the other hand, several requirements are domain independent. For such requirements the implementation in a software could be done with **design patterns**. Such patterns can help to reduce the time effort for the realization of a requirement in a knowledge engineering process. For example, a notebook recommendation system contains products, questions to customers, and relationships between products and customers (constraints). In this domain products have different prices and customers will be asked for their maximum price. While the product variable *price* may have hundreds of different prices (domain elements), the customer will not choose, e.g., between $price = 799.90$ or $price = 799.99$ but wants to have, for example, ten different prices (e.g. $price \leq 400$ or $price \leq 600$ or ... or $price \leq 2200$). The relationship between those variables can be denoted as *mapping* which could be a design pattern.

5 Intelligent Supporting Techniques for Maintaining Constraint-based Systems

We have presented several techniques to support knowledge engineers and customers when dealing with product configuration systems. In the next chapter we present interfaces for knowledge engineers and end-users focusing on the abilities and capabilities of those systems and introduce our new system iCone.

6 Interfaces for the Maintenance of Constraint-based Systems

In this Chapter we focus on visualization techniques for anomaly management. Therefore we first analyze several principals for configuration systems and present selected available configuration systems in Section 6.1. 6.2 introduces a system for the development and maintenance of constraint-based product configuration systems.

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems¹

Configuration is one of the key enabling technologies of Mass Customization [Pine and Davis, 1999; Anderson, 1997]. It can be defined as a special case of *design activity* where the resulting artifact is composed of a predefined set of component types and is consistent with a given set of constraints [Sabin and Weigel, 1998]. Configuration environments allow *end-users* to design their own individualized products and services as well as *knowledge engineers and domain experts* to develop configurator applications (mainly user interfaces and knowledge bases). For both types of users the user interface of the configuration environment plays a key role. *First*, a major precondition for *lasting acceptance and application* is that end users can easily configure a product that fits their wishes and needs. *Second*, knowledge engineers and domain experts are in the need of technologies that allow

¹This Section is based on Felfernig et al. [2014a].

6 Interfaces for the Maintenance of Constraint-based Systems

effective development and maintenance processes which are a precondition for *up-to-date knowledge bases*.

The major contributions of this section are the following: we provide an overview of important *criteria* to be taken into account when we develop user interfaces for configurator applications as well as interfaces for the corresponding knowledge engineering environments. For the discussed criteria we present and exemplify technologies which help to take these criteria into account. With the goal to stimulate further research on user interfaces for configurator environments, we discuss relevant issues for future work. A summary concludes this section.

Design Principles for Configuration Interfaces

Contrary to conventional product design where experienced product designers and managers are responsible for the design of product alternatives offered to a customer, the task of customization is now also forwarded to *end users*. An important aspect in this context is that we can not expect the disposal of detailed technical product domain knowledge from end users because they typically do not know their preferences [Simonson, 2003]. As has been analyzed by [Randall et al., 2005], there are problems with existing configurator user interfaces which have to be tackled in order to increase the usability of configuration technologies. We now summarize *five key-principles* to be taken into account when developing user interfaces for configuration environments [Randall et al., 2005]. In this context we focus on both, interfaces for *users of a configurator application* as well as interfaces which support *knowledge engineers and domain experts* in configurator application development and maintenance.

Principle 1: Customize the Customization Process

Sales persons are typically adapting their style of customer interaction depending on the type of customer. For example, customers interested in

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

technical details should be supported by in-depth technical information and corresponding analyses. In contrast, customers without expertise in the product domain should be supported by a more function-oriented dialog where technical details are omitted. Since we can expect different types of end users of a configurator application, we are in the need of technical approaches that help to personalize the interaction with the configuration environment depending on the basic characteristics of the current user. As proposed by [Randall et al., 2005], a configuration environment should at least support two different types of user interfaces: a *needs-based interface* for non experts and a *parameter-based* one for users who want to perform configuration primarily on the basis of the given set of technical product properties.

Similar to end users there are also different types of *knowledge engineers and domain experts* responsible for the development and maintenance of a configurator application, e.g., their knowledge elicitation, data analysis and knowledge representation methods can differ [Shortliffe and Patel, 2007]. There may be experts who know every technical property of the product but there are also employees who recently started working on the configurator application. These two types of users are in the need of different navigation support.

Principle 2: Provide Starting Points

Users of a configurator application can not only be differentiated with regard to their product preferences [Domshlak et al., 2011]. They can also be differentiated with regard to their *preferred product properties*, i.e., properties of a product they are interested in and would like to specify. For example, one user of a car configurator prefers to start with a specification of the basic *car type* (e.g., *limousine vs. combi*) whereas another user prefers to specify the *price* because this parameter has the highest priority. In this context, a basic requirement for configurator user interfaces is the provision of so-called *starting points* which can be seen as a kind of initial variable assignment from which the customer can continue the configuration activities [Mandl et al.,

6 Interfaces for the Maintenance of Constraint-based Systems

2011b]. The important idea behind starting points is that not every user is interested in designing (configuring) a product from scratch but rather relies on existing basic settings (also denoted as *stereotype configurations*).

Starting points should be provided for *knowledge engineers as well as for domain experts*. A knowledge engineer (or domain expert) with nearly no knowledge about an already existing configuration knowledge base is clearly in the need of an indicator of where to best start the analysis process. Such starting points can improve the overall efficiency of developing a basic understanding of a knowledge base – or more general – a configurator application.

Another example for a starting point in the context of knowledge base development and maintenance are recommendations for the maintenance derived from a quality analysis of the knowledge base [Felfernig et al., 2009, 2011c].

Principle 3: Support Incremental Refinement

Users of a configurator application typically construct their preferences within the scope of a configuration session [Simonson, 2003], i.e., preferences are not known beforehand. A typical requirement for user interfaces in this context is that the configurator should actively support *sensitivity analysis* in the sense that tradeoffs between different properties are visualized and alternatives to the current configuration are shown in an intuitive fashion. An easy way to support such a tradeoff analysis is to include *product comparison* functionalities, i.e., a comparison between different alternative configurations. An important functionality in this context is that the configurator is able to *automatically* determine alternative configurations on the basis of defined user preferences.

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

Similarly, there is a need for the support of *tradeoff analysis in the context of knowledge base development and maintenance*. In this context, a knowledge engineer (or domain expert) should be supported in correction and extension activities. If, for example a domain expert has specified examples for the intended behavior of a configurator in terms of which configuration should be shown in which context and the configurator is not able to fulfill these requirements, the domain expert should be supported in terms of indications of the sources of the given inconsistency [Felfernig et al., 2004b]. In this context the user interface should support the comparison of explanations for the inconsistencies and it should provide information about the pragmatics of changes to the knowledge base, for example, in terms of test cases which are additionally accepted by the knowledge base due to the performed repair operations.

Principle 4: Exploit Prototypes to Avoid Surprises

A major obstacle for the willingness to purchase a product is missing *trust* in a given configuration [Grabner-Kraeuter and Kaluscha, 2003]. Since configurations in many cases are unique, the exploitation of product evaluations for decision making is often impossible. Consequently, alternative concepts have to be provided which help the customer to anticipate the post-purchase experience [Randall et al., 2005]. In this context it is important to visualize the impact of different decision alternatives on the final configuration outcome. For example, a change of the *interieur color* of a car has to be immediately shown to the user by directly visualizing the new interieur of the selected *car type*. Another example is the configuration of financial service portfolios: after having increased the level of *willingness to take risks* or having increased the *expected yearly return rate* of a financial service, the configurator should immediately show the possible consequences of the taken decision, for example, in terms of *less available money* due to plunging *stock prices* or – on a more visual level – in terms of the possibility of not being able to buy the envisioned car.

6 Interfaces for the Maintenance of Constraint-based Systems

In the context of *configurator application development*, we are as well in the need of prototypes that visualize the impact of changes in products (e.g. a product is not selectable) and constraints (e.g., a constraint is redundant). When, for example developing a knowledge base, knowledge engineers and domain experts should immediately be able to test the new version of the knowledge base by interacting with the configurator, i.e., new versions of a knowledge base have to be automatically integrated with the corresponding user interface. Another example of a visualization in the context of knowledge base development and maintenance is to show information regarding quality properties of the knowledge base. If a knowledge engineer adapts or extends the knowledge base, feedback should be given immediately, for example, in terms of an indication of a deterioration or improvement of the level of maintainability.

Principle 5: Teach the Consumer

Consumers very often do not have the technical background knowledge of important technical properties of the product, for example, they do not know which processor type has which performance and – even more important – they do not know which performance is needed for their specific requirements [Randall et al., 2005]. Increasing knowledge of the properties of a configuration can also increase the corresponding *willingness to buy* [Felfernig et al., 2006b]. As a consequence we are in the need of mechanisms that assist users and help them to increase their personal product domain knowledge or reduce the effort for taking decisions. Besides basic concepts such as *help buttons* that explain the role of specific product parameters, more sophisticated explanation functionalities can be integrated into configuration environments. *First*, users are typically in need of explanations as to why a concrete configuration has been recommended. This functionality can simply be supported by interpreting the information received from the configuration environment (i.e., which constraints were responsible for the calculation of the configuration). *Second*, users are also in the need of explanations in situations where no solution can be found by the configuration system. In this context the user is in the need of information which

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

requirements are responsible and what are possible repair actions that allow the calculation of at least one solution.

In the context of *configuration knowledge acquisition and maintenance*, similar explanation functionalities are needed: knowledge engineers and domain experts need to be informed about the sources of inconsistencies in the knowledge base (in the case that some of the test cases fail) and what are the corresponding alternatives for repairing the knowledge base. Another type of explanation is stemming from collaborative recommendation [Konstan et al., 1997] where explanations are typically presented in the form of *users who purchased item A also purchased item B*. This type of explanation can as well be applied in the context of knowledge base development and maintenance, for example, in the form of *If constraint A has changed you should also change constraint B*.

Using the Design Principles in Configuration Environments

On the basis of our discussion of *basic design principles* for configurator user interfaces in the previous section, we now focus on a discussion of *basic technologies* that can help to support these design principles. In the following we show how the principles can be used to optimize user interfaces for potential customers. Thereafter we present knowledge acquisition environments which are using the design principles.

Customize the Customization Process. User interfaces of configurator applications can simply be implemented on the basis of explicit definitions of a configuration process. Such a process defines in which order which parameters have to be specified by the user. For our computer configuration example [Felfernig et al., 2014c] we can specify, for example, two different types of user interfaces. For *experts* in the domain of personal computers, we could provide a search interface where a user (customer) can specify the requirements on the level of technical product properties such as *type of cpu, motherboard, operating system, screen, hd unit, internet, and application*

6 Interfaces for the Maintenance of Constraint-based Systems

software. For *non-experts* in the product domain, a user interface could pose questions on a more abstract (needs-oriented) level such as primary types of usage (e.g., *multi-media & game playing, programming, complex mathematical calculations, and office applications*). This needs-oriented view on the product assortment is often denoted as *functional architecture* of a configurable product [Felfernig et al., 2014c]. From the specification of preferences, the configurator application can then determine the corresponding technical properties. For example, *programming and complex mathematical applications require a multi-core cpu architecture*.

On both levels (functional architectures and detailed technical product properties) questions posed to the customer have to be ranked. The simplest way to achieve this is to specify a static ordering where, for example, the question regarding the *type of cpu* is posed first. However, more flexible approaches have been developed which help to *adapt their strategy regarding the selection of the next question* depending on the navigation behavior of the user. One approach to support a selection of questions *on the fly* is to apply the concepts of collaborative filtering [Falkner et al., 2011; Konstan et al., 1997]. The idea of collaborative filtering is to determine recommendations for the current user on the basis of the preferences of users with a similar navigation behavior as the current user. In our scenario – in order to determine the next question to ask the current user – we would determine users with a similar navigation behavior (the *nearest neighbors*) and on the basis of the preferences of the nearest neighbors try to select (recommend) the next question to be posed to the current user.

A working example for such a collaborative dialog management is shown in Table 6.1. The users 1..5 have already completed their configuration sessions, for example, user 1 has first specified a value for *CPU*, then a value for *OS*, then a value for *HD*, thereafter a value for *motherboard*, then a value for *Screen*, *Internet*, and *Applications*. The *current user* has already specified values for the parameters *CPU* and *OS* and we would like to predict which of the remaining parameters will be specified by the current user next. The nearest neighbors of the current user are *user1*, *user3*, *user 4* since all of them have first selected a *CPU* and afterwards selected an operating system (*OS*).

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

<i>user</i>	<i>CPU</i>	<i>motherboard</i>	<i>OS</i>	<i>Screen</i>	<i>HD</i>	<i>Internet</i>	<i>Applications</i>
1	1	4	2	5	3	6	7
2	3	1	2	4	5	7	6
3	1	3	2	5	4	7	6
4	1	3	2	4	5	6	7
5	2	1	3	5	4	6	7
current	1	?	2	?	?	?	?

Table 6.1: Example of determining relevant questions on the basis of *collaborative filtering* [Konstan et al., 1997].

Since the majority of nearest neighbors has selected *motherboard* as third parameter, we recommend *motherboard* as next parameter to be specified by the current user. For a more detailed overview of *parameter selection methods* we refer the reader to [Falkner et al., 2011].

Provide Starting Points. The basic approach to the provision of starting points is to predetermine parameter settings which are of interest for the user. Such starting points are often denoted as *default values* which are proposed to the user within the scope of a configuration session. On the one hand defaults can represent specific parameter settings, for example, *in german-speaking countries the value for the keyboard part of the configuration is set of german*. On the other hand, defaults can represent whole sub-configurations, for example, *the default installation settings for a software package included in a computer configuration*. Independent of the generic type of default (i.e., specific parameter setting vs. whole sub-configuration), defaults can be distinguished in terms of the way that default values are determined. Three basic types of defaults will be discussed in the following.

Static Defaults. A parameter has a fixed predefined default value which is completely independent of the current configuration context. For example, the default value of the parameter *internet* is set to *yes* since it is assumed that most of the users want to have included the corresponding network hardware. The application of this type of default is limited since

6 Interfaces for the Maintenance of Constraint-based Systems

in many cases defaults depend on the context of user interaction. This aspect of *contextualization* is taken into account by the following two types of defaults.

Rule-based Defaults. Here the determination of defaults is personalized and the selection of default values is performed on the basis of pre-defined default rules. An example for such a rule is the following: *if the user has selected 'programming' as a main field of application then the value of the parameter 'memory' must be set to 4GB.* Although this default type takes into account context information (on the basis of rules), rules trigger knowledge acquisition and maintenance efforts. The following default type does not rely on an explicit specification of contexts but on context learning on the basis of an analysis of already completed configuration sessions.

Adaptive Defaults. If we want to keep knowledge base development and maintenance efforts as low as possible, we have to develop mechanisms which are able to automatically determine parameter settings of relevance for the current user. Different types of machine learning approaches can be applied to support such an adaptive determination of defaults [Falkner et al., 2011; Tiihonen and Felfernig, 2010]. An example for the application of collaborative filtering is given in Table 6.2. The default value for the parameter *OS* (the operating system) proposed to the *current user* would be *OS-Alpha* since the nearest neighbors of the *current user* (the users 1 and 5) have chosen *OS-Alpha*. An example user interface - which is based on adaptive default values - is depicted in Figure 6.1. This configurator (RECOMOBILE) supports the configuration of mobile phones (for details see [Felfernig et al., 2014c]).

Support Incremental Refinement. An important feature to support a user in preference construction is to explicitly visualize the existing tradeoffs between different configuration alternatives in corresponding comparison tables [Felfernig et al., 2006b]. An often used approach to support such a tradeoff analysis is to simply rank alternative configurations with regard to their *price* (see Figure 6.2). Presenting configuration alternatives in such

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

<i>user</i>	<i>CPU</i>	<i>motherboard</i>	<i>OS</i>	<i>Screen</i>	<i>HD</i>	<i>I-net</i>
1	CPU-D	MB-Diamond	OS-Alpha	Screen-A	MedStore	Yes
2	CPU-S	MB-Silver	OS-Beta	Screen-B	MedStore	No
3	CPU-S	MB-Silver	OS-Beta	Screen-A	MaxStore	Yes
4	CPU-S	MB-Silver	OS-Beta	Screen-B	MedStore	No
5	CPU-D	MB-Diamond	OS-Alpha	Screen-A	MedStore	No
curr.	CPU-D	MB-Diamond	?	?	?	?

Table 6.2: Example of determining default values for the parameters of the computer configurator on the basis of *collaborative filtering* [Konstan et al., 1997].

a way *biases* the product comparison towards an evaluation of the price attribute [Felfernig et al., 2014c]. Since *price* is in many cases the most important decision criterion, such interfaces are often used in commercial settings. The major drawback of this type of product comparison is that the interface in no way takes into account the real preferences of the user.

If user preference information is available, it can be taken into account when presenting configuration alternatives. For the purpose of our example let us assume that the preferences of the current user regarding the importance of the different product properties is known (see Table 6.3). Furthermore, we dispose of a utility evaluation of the different product properties (see Table 6.4).

<i>Price</i>	<i>CPU</i>	<i>Motherboard</i>	<i>Internet</i>
5%	35%	35%	20%

Table 6.3: Example list of user preferences.

On the basis of this information and a corresponding utility function we can determine the user-specific utility of each configuration alternative x [Winterfeldt and Edwards, 1986]. The utility function used for the purposes of our example is shown in Formula 6.1 where x denotes a specific configuration alternative, $importance(i)$ denotes the importance of product

6 Interfaces for the Maintenance of Constraint-based Systems

The screenshot shows the RecoMobile interface with a progress bar at the top indicating steps: Needs, Subscription, Privacy, and Phone. The 'Phone' step is currently active. Below the progress bar, there is a note: "The preselected values are recommended. You can either change or accept them (with the corresponding Accept-Button). To accept all recommendations at once use the 'Accept all recommendations' button." The main content consists of five questions with radio button options and a small envelope icon next to the selected option:

- Do you want to use your mobile phone to read/write Emails?
 - No
 - Occasionally
 - Daily
 - All the time
- Do you want to use your phone to connect your PC to Internet?
 - No
 - Occasionally
 - Frequently
- Do you want to send SMS?
 - No
 - Occasionally
 - Daily
 - Several per Day
- Do you want to use your phone for sports tracking?
 - No
 - Yes
- Do you want to use your phone for GPS navigation?
 - No
 - Yes

At the bottom right, there are three buttons: "ACCEPT ALL RECOMMENDATIONS" (with a checkmark icon), "<< BACK", and "NEXT >>". Below these buttons is a link: "View your selections".

Figure 6.1: RECOMOBILE user interface for the representation of (adaptive) defaults.

property i for the current customer, and $contribution(x, i)$ denotes the utility of configuration x with regard to property i .

$$utility(x) = \sum_i^n importance(i) \times contribution(x, i) \quad (6.1)$$

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

Price	\$389	\$455	\$612
CPU	CPU-S	CPU-S	CPU-D
Motherboard	MB-Silver	MB-Silver	MB-Diamond
...
Internet	No	Yes	Yes

Figure 6.2: Configuration comparison interface based on price ranking.

Attribute	Value	Utility
CPU	CPU-S	4
	CPU-D	7
Motherboard	MB-Diamond	8
	MB-Silver	2
Internet	Yes	10
	No	1
Price	0–400	10
	401–600	7
	601–1000	3

Table 6.4: Example list of product utilities.

Combining the customer-specific preferences (Table 6.3) with the attribute utilities defined in Table 6.4, we receive the ranking $configuration(\$612) > configuration(\$455) > configuration(\$389)$. This shows that the most expensive configuration can also have the highest utility for a user. A corresponding comparison interface is sketched in Figure 6.3.

Note that tradeoff analysis does not only play a role in the context of comparing different solution alternatives with regard to their utility for the user. In situations where no solution can be found for the current set of user requirements (specified preferences), the configurator application proposes a set of repair alternatives, i.e., alternatives for changes to the current set of requirements that can guarantee the retrieval of a solution [Felfernig

6 Interfaces for the Maintenance of Constraint-based Systems

	FirstClass	Business	Economy
CPU	CPU-D	CPU-S	CPU-S
Motherboard	MB-Diamond	MB-Silver	MB-Silver
...
Internet	Yes	Yes	No
Price	\$612	\$455	\$389

Buy this!

Buy this!

Buy this!

Figure 6.3: Configuration comparison interface based on utility-based ranking.

et al., 2009]. Such repair alternatives can be represented in the same way as alternative configurations (see Figure 6.4).

Repair(#Changes)	Ω_1 (1)	Ω_2 (1)
CPU	CPU-D	CPU-S (CPU-D)
Motherboard	MB-Diamond	MB-Silver (MB-Diamond)
Price	\$612 (\$390)	\$390
Internet	Yes	No (yes)

Accept Changes!

Accept Changes!

Figure 6.4: Configuration comparison interface based on utility-based ranking.

Exploit Prototypes to Avoid Surprises. Especially for online configuration environments it is crucial to take the aspect of *trust* into account since this factor is one of the most important when it comes to a purchase decision [Felfernig et al., 2006b; Grabner-Kraeuter and Kaluscha, 2003]. Due to large solution spaces typically specified by configuration knowledge bases, the resulting configurations are in many cases unique. As a consequence, product evaluations are not available. All the more, a concrete visualization of the outlook (and pragmatic) of the configuration is crucial in order to give the customer a clear impression of a potential post-purchase experience.

Teach the Consumer. The basic and wide-spread approach to support a better understanding of configuration results as well as situations where

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

inconsistencies occur, is to provide *explanations* (see Section 4.2). Basically, we can differentiate between *two* basic forms of explanations. *First*, an explanation can provide a set of *argumentations* as to why a configuration alternative has been recommended. *Second*, in situations where no solution can be found by the configuration environment, the user has to be informed about the underlying inconsistencies and potential repairs (see Section 4.4).

Beside these basic types of explanations, all the criteria for the development of configurator user interfaces discussed in the prior sections also contribute to a better understanding of the product domain and the underlying configuration knowledge bases. *Customizing the customization process* allows users to grapple with topics they are interested in and – as a consequence – to know more about relevant concepts of the product domain.

Technological Issues

On the basis of our discussion of *basic design principles* for configurator user interfaces in the previous section, we now focus on a discussion of basic technologies that can help to support these design principles for end user interfaces and user interfaces for knowledge engineers.

Customize the Customization Process. For using the customizable process [Blythe et al., 2001b] developed a knowledge base environment with the ability to support domain experts inserting new knowledge into the KB. Considering the side effects of the changes, the authors developed an approach to inform the user about these side effects via a customized tunneling approach.

6 Interfaces for the Maintenance of Constraint-based Systems

<i>user</i>	c_1	c_2	c_3	c_4	c_5	c_6
1	4	2	3	5	1	6
2	3	2	5	6	1	4
3	1	3	2	4	6	5
4	3	2	4	5	1	6
current	?	2	?	?	1	?

Table 6.5: Example of determining relevant constraints on the basis of *collaborative filtering* [Konstan et al., 1997].

A similar approach can support the navigation of knowledge engineers (domain experts) in complex knowledge bases (see the example in Table 6.5). In our scenario, the knowledge engineers (users 1..4) have already interacted with the knowledge base, for example, user 1 took a look at all the constraints in the following order: $c_5, c_2, c_3, c_1, c_4, c_6$.² The *current user* has already inspected the constraints c_5 and c_2 . The constraint which should be recommended next to the current user is c_1 since this one has been inspected next by the majority of the nearest neighbors of the current user (user 2 and user 4).

The ICONE knowledge acquisition prototype is an example of a graphical knowledge acquisition user interface that includes *recommendation technologies* for proactively supporting knowledge engineers and domain experts in their development and maintenance activities. A detailed description of the application is given in Section 6.2.

Provide Starting Points. In the context of *knowledge engineering and maintenance scenarios*, defaults play a major role as well. The recommendation of constraints which could be of relevance for the current user (knowledge engineer or domain expert) can be interpreted as a specific type of adaptive default. Similar defaults can be determined for diagnoses [Felfernig and Schubert, 2011a] or sets of redundant constraints, i.e., constraints which do

²Note that for reasons of simplicity we assume that each stakeholder has inspected each constraint at least once.

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

not change the semantics of the knowledge base when deleted [Felfernig et al., 2011c]. In this context a default can be interpreted as a set of faulty or redundant constraints.

Another type of providing a starting point is to structure the user interface in a way that only simple constraints can be inserted. By limiting the interface to the implication constraint, we can use the advantages researched in [Felfernig et al., 2010] and reduce the cognitive effort of the user interface. If more complex constraints are absolutely necessary, the knowledge engineer has the possibility to insert not limited constraints.

Support Incremental Refinement. Tradeoff analysis in the context of *configuration knowledge base development and maintenance* has a similar need in terms of user support. When testing a knowledge base, knowledge engineers have to figure out which constraints are responsible for the faulty behavior of a knowledge base (e.g., the knowledge base calculates configurations which are not feasible on the technical level) and which are the repair alternatives to be taken into account (a detailed discussion is given in Chapter 4). In this situation well different subsets of constraints have to be evaluated with regard to the probability of being responsible for the faulty behavior of the knowledge base. A detailed technical discussion of the techniques supporting the automated identification and ranking of faulty constraint sets can be found in [Felfernig et al., 2014c].

Exploit Prototypes to Avoid Surprises. Prototyping concepts also play a major role in the context of *configurator application development and maintenance*. When creating or adapting a configurator application, the knowledge engineer should be able to immediately analyze the impact of changes on the layout of the configurator interface as well as on the underlying configuration logic. The configuration environment COMBEENATION³ is an innovative *look and feel* environment that supports application development and maintenance processes on a graphical level (see Figure 6.5). The

³See www.combeeneration.com.

6 Interfaces for the Maintenance of Constraint-based Systems

definition of a configuration knowledge base is product-centered which means that component and constraint definitions are directly attached to a graphical representation of the configurable product. This approach allows knowledge-based system development and maintenance for *rapid prototyping processes*. Furthermore, COMBEENATION allows an immediate user testing and feedback, i.e., erroneous behavior of the application can be immediately reported to the responsible knowledge engineer. Further research related to graphical development, testing, and debugging environments for configurator applications can be found in [Felfernig, 2007].



Figure 6.5: COMBEENATION: integrated development and visualization of configurators.

Teach the Consumer. In the context of *knowledge base development and maintenance*, knowledge engineers and domain experts are in a similar situation – they have to be informed about the sources of faulty behavior,

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

for example, in the case that certain test cases become invalid due to a prior change to the knowledge base. In such situations, diagnoses take over the role of explanations [Felfernig et al., 2014c] (see Section 4.2).

Similarly, knowledge engineers can efficiently develop a basic understanding of the most relevant components and constraints part of the configuration knowledge base. On the one hand, the *provision of starting points* offers initial and reasonable settings in terms of, for example, a partial configuration and thus helps to reduce overheads related to the design of consistent configurations. On the other hand, starting points make knowledge engineering operations more efficient due to the availability of additional indicators of potential sources of inconsistencies. *Knowledge engineers and domain experts* are supported in developing a basic understanding of the tradeoffs with regard to alternative repair operations needed to restore the consistency of a knowledge base. Supporting the concept of *incremental refinement*, for example, on the basis of product comparison pages, helps the user to develop a clear understanding of existing tradeoffs in the space of solutions offered by the configuration knowledge base. Finally, *prototypes* that are visualizing alternative configurations can significantly help to develop a better understanding and clearer preferences regarding certain solution alternatives. In the context of knowledge base development, such visualizations help to understand alternative impacts of change operations on the next version of the knowledge base [Felfernig, 2007].

In order to summarize the discussed design principles for user interfaces of configuration environments, we provide an overview of these principles which includes related technological foundations (see Table 6.6).

6 Interfaces for the Maintenance of Constraint-based Systems

Principle	Technological Foundations
Customize the Customization Process	Parameter Selection [Falkner et al., 2011]
	Adaptive Knowledge Acquisition [Burke et al., 2011]
Provide Starting Points	Recommendation of Defaults [Falkner et al., 2011; Tiihonen and Felfernig, 2010]
	Recommendation of Knowledge Base Diagnoses [Felfernig and Schubert, 2011a]
Support Incremental Refinement	Configuration Comparison [Felfernig et al., 2006b] [Felfernig et al., 2006b]
	Diagnosis Comparison [Felfernig et al., 2009]
Exploit Prototypes to Avoid Surprises	Explanations [Felfernig et al., 2006b] Graphical Testing & Debugging [Felfernig, 2007]
Teach the Consumer	Diagnoses [Felfernig et al., 2011a]
	Explanations [Friedrich and Zanker, 2011]
	Recommendation [Felfernig and Schubert, 2011a]

Table 6.6: Design principles of configurator user interfaces and technological foundations.

Research Issues

The diversity of the configuration model has a strong influence on the design and structure of user interfaces – nevertheless certain commonalities can be identified. The *Configuration Database Project*⁴ provides a collection of online configurators and includes more than 800 entries. After having analyzed configuration environments contained in this database, we detected that more than 50% of the analyzed configurators show the following basic characteristics:

- Selected components are summarized at the end of the configuration process.
- Products available for configuration are presented as images.
- Process navigation, if available, is structured on a horizontal plane.

⁴See www.configurator-database.com

6.1 Current user Interfaces for the Maintenance of Constraint-based Systems

- Choice fields are positioned next to and/or beneath the product picture.
- Shopping cart, order button, and total price are clearly visible.

A configurator application that takes into account the mentioned characteristics is depicted in Figure 6.6. It is intended as an example for developers of configurator applications. Important hints regarding the design of individual configurator user interface elements are the following:

- The logo should be shown in a dominant position for a fast identification.
- The navigation bar should be clearly visible and shown unfragmented.
- The size of product (configuration) images should be sufficient to see details.
- Selection box(es) should follow a logical clustering.
- Prices (price tables) should be accessible in all steps of a session.
- User preferences should be adaptable (e.g., by back/forward navigation).
- Shopping cart and order-button should be available for completion purposes.

Summarizing, a *close-to-reality* approach is recommended to reduce the uncertainties that a customer feels regarding a product that is not tangible yet. For a more detailed discussion on usability issues in configurator user interface development we refer to [Rogoll and Piller, 2004].

Conclusions

With this section we provide an overview of relevant principles of developing user interfaces for configuration environments. In this context we focused on both types of user interfaces, *interfaces for the end-user* and *interfaces for knowledge engineers and domain experts* who are in charge of

6 Interfaces for the Maintenance of Constraint-based Systems

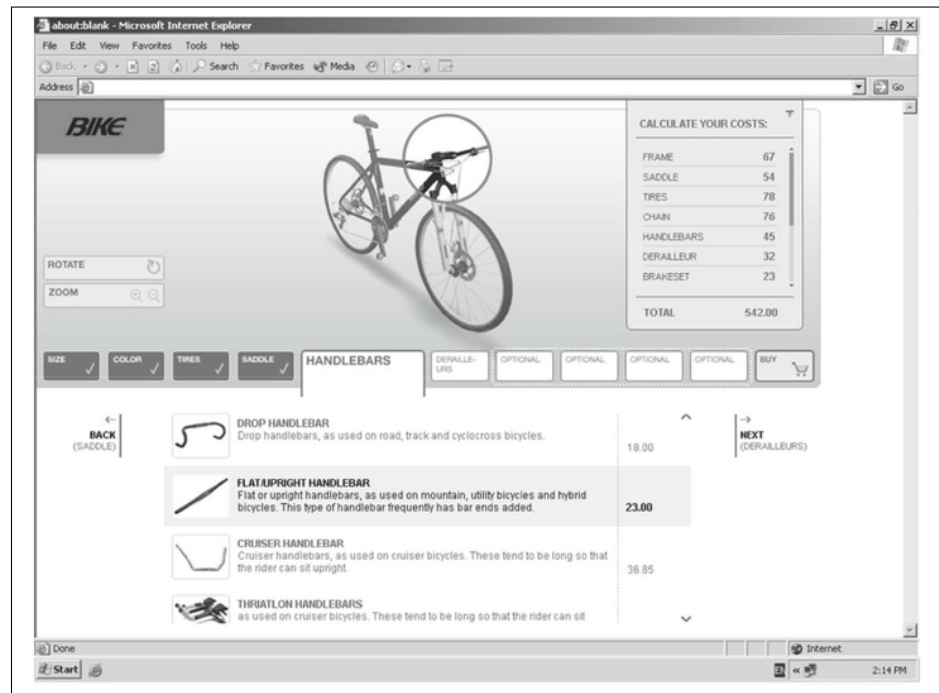


Figure 6.6: A prototype web-based bicycle configurator (see www.cyledge.com).

knowledge base development and maintenance. In addition to this discussion of design principles, we proposed a couple of technologies which help to achieve the mentioned principles. How we have realized the design principles, will be shown in our interface implementation in the following Section.

6.2 iCone: an Interface for the Maintenance of Constraint-based Systems⁵

In this Section we give an overview of the iCone-interface and intelligent techniques like recommendation, anomaly management, dependency detection, and metrics which are implemented in the system.

⁵This Section is based on [Wotawa et al. \[2015a\]](#).

6.2 iCone: an Interface for the Maintenance of Constraint-based Systems



Figure 6.7: Main screen of the icone interface

iCone ('Intelligent Environment for the Development and Maintenance of Configuration Knowledge-Bases') is a java-based web-application.⁶ Figure 6.7 shows the home screen of the application.

At the top-right of the page is the *login area*. The area on the left side of the page shows the *navigation*. On the right side of the page is the *recommendation and notification area*. In Figure 6.7 the user sees a list of recommended knowledge bases ('Notebook' and 'Smartphone'). A detailed description of recommendations and notifications is given in the following section. Finally, the main area of each page is the *content area*. For example, in Figure 6.7 the user sees a list of knowledge bases where the user is invited.

An overview of the principles of the iCone system can be found in Figure 1.1. The main object of iCone is the *knowledge base*. It contains all products, product variables, questions, and constraints. Furthermore, it deals with the

⁶<http://ase-projects.studies.ist.tugraz.at:8080/iCone>

6 Interfaces for the Maintenance of Constraint-based Systems

analysis package which detects anomalies in the knowledge base, generates recommendations for the knowledge engineers, approximates dependencies between variables in the knowledge base, and generates metrics to evaluate the knowledge base. For some evaluations of a knowledge base we need consistency checks which will be done by *SQL statements*. SQLite databases are used to do consistency checks and to save and load knowledge bases. Finally, the iCone *interface* interacts with knowledge engineers. The interface can be used to develop and maintain knowledge bases and to get a preview for the resulting recommendation / configuration knowledge base.

Recommendation Techniques

Recommendation techniques help knowledge engineers to focus on relevant items (e.g. products, questions, and constraints) in the knowledge base. For example, a knowledge engineer focuses on relevant constraints for the customer variable *usage_scenario*. In that case recommendation techniques find relationships between constraints with the corresponding variable. We implemented several types of recommendation techniques. Figure 6.8 shows the presentation of the implemented recommendation techniques.

We implemented four different recommendation techniques which can be divided into user-independent and user-dependent ones. A detailed description is given in Section 5.1.

- user-independent recommendation techniques
 - most viewed: this recommendation technique shows the most viewed items. It will not be differed if the item is viewed or edited.
 - recently added: this recommendation gives a list of the newest items in the knowledge base.
- user-dependent recommendation techniques

6.2 iCone: an Interface for the Maintenance of Constraint-based Systems

The screenshot displays the iCone web interface. At the top, the logo 'iCone' is followed by the tagline 'Intelligent Environment for the Development and Maintenance of Configuration Knowledge Bases'. A user login status 'You are logged in as freinfra, logout' is visible in the top right. Below the header is a navigation menu with options: Home, Laptop, Viewer, and iCone-Admin. The main content area is divided into several tabs: Requirements, Exclusions, Dependencies, Incompatibilities, Logical constraints, Variable mappings, and Self defined. The 'Requirements' tab is active, showing a table with columns for ID, Name, and Content. The table contains three rows of constraints, each with a 'verify' button. Below the table is an 'Add filter:' input field. On the right side of the interface, there are three recommendation sections: 'Recommended to you', 'Most viewed ...', and 'Recently added', each displaying a list of constraint IDs. A 'Notifications' section at the bottom right shows a system message: '2015 / 4 / 27 / 11:11 SYSTEM: User Frank Witzama is also working on the same knowledge base 'Laptop' (version 4)'. The footer contains copyright information for the Institute for Software Technology (2013), version v1.0.3 (2015 / 04 / 15), and a 'Contact and Impressum' link.

Figure 6.8: Presentation of constraints and recommendation techniques. The right area contains the collaborative filtering ('Recommended to you'), the most-viewed ('Most viewed ...'), and the newest items ('Recently added') recommendation engine. If the user clicks on the 'i'-symbol in the content area which is available for each constraint, question, and product in the system, she receives the recommendations from the content-based filtering recommendation.

- collaborative filtering: this recommendation technique finds relationships between knowledge engineers based on their most viewed and edited items. For example, if user *A* takes a look for the product variable *cpu_cores* and the customers *usage_scenario* and user *B* is focusing on items concerning the *cpu_cores* product variable, she might also be interested in the question variable *usage_scenario*.
- content-based filtering: based on the content (variables and domain elements) of products, questions, and constraints the system exploits the content of products (C_P), questions (C_R), and constraints (C_{KB}) to find similar items in the knowledge base. For example, if a constraint is concerned to the variable *usage_scenario*, other constraints with the same variable will be shown.

6 Interfaces for the Maintenance of Constraint-based Systems

For a detailed explanation of the different types of recommendation techniques and empirical studies, we refer the reader to Section 5.1 and [Reinfrank et al., 2015a; Felfernig et al., 2013b; Jannach et al., 2010; Chen and Pu, 2006; Burke, 2000].

Anomaly management

Anomalies are patterns in data that do not conform to a well defined notion of normal behavior [Chandola et al., 2009]. For example, in Figure 6.9 are two sets of well-formedness violations.

In our implementation we consider three different types of anomalies: conflicts, redundancies, and well-formedness violations.

- **conflict:** A conflict is a set of constraints which can not be fulfilled. For example, a constraint $price < 599 \wedge usage_scenario = gaming$ can not be fulfilled because *gaming* implies four *cpu_cores* and the notebooks with four *cpu_cores* costs at least 599 EUR. Conflicts can be resolved by sets of **diagnoses**. A diagnosis contains a set of constraints. If all constraints, which are part of a diagnosis, are removed from a knowledge base, the knowledge base will be consistent. A detailed discussion is given in Section 4.1.1.
- **redundancy:** A set of constraints can be denoted as redundant if the removal of this set does not change the behavior of the knowledge base. For example, a constraint $price < 700$ can be removed without changing the semantics if there exists another constraint $price < 500$ (see Section 4.1.2).
- **well-formedness violation:** Well-formedness violations do not change the behavior of the knowledge base but make it difficult to maintain a knowledge base. Well-formedness violations subsume different types of violations like '*dead domain element*' (e.g., if a variable can never have a specific value), '*full mandatory*' (if variables must have a specific assignment), and '*unnecessary refinement*' (if each possible value of a variable is always combined with another domain element of another

6.2 iCone: an Interface for the Maintenance of Constraint-based Systems

domain value). Figure 6.9 shows that our example knowledge base has two unnecessary refinements. For example, the variable *usage_scenario* can be replaced by *cpu_cores* because each time when the customer wants a notebook for *gaming* she receives 4 *cpu_cores* and when she wants a notebook for *multimedia* or *office* she receives a notebook with 2 *cpu_cores*. A technical description of well-formedness violations is given in Section 4.1.3.

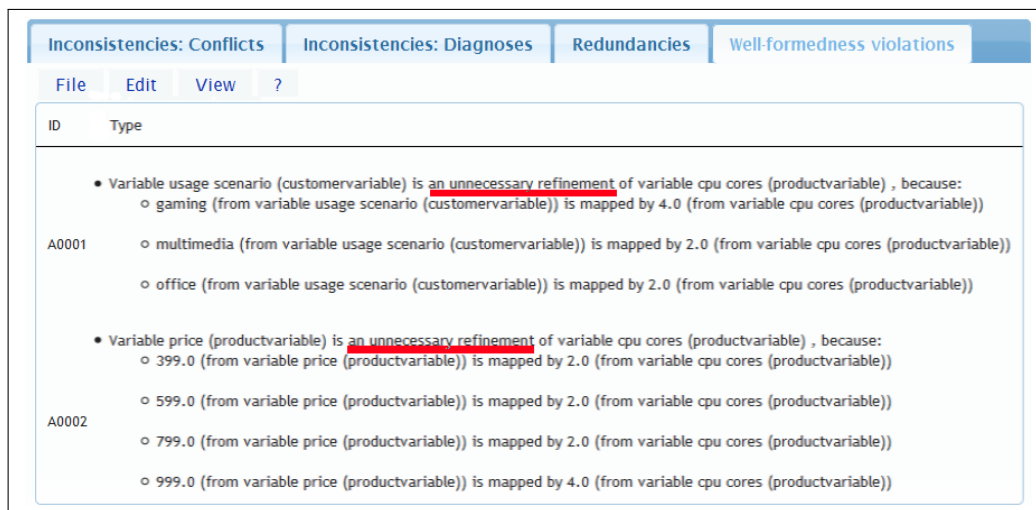


Figure 6.9: Presentation of anomalies in the iCone interface: with the tabs at the top of the main content area the user can switch between conflicts, diagnoses, redundancies, and well-formedness violations. Our example knowledge base does not contain conflicts, diagnoses, and redundancies but contains two unnecessary refinements.

For each anomaly the system offers a mail draft and a forum. Additionally, the system offers links to delete elements of anomalies (e.g. delete a constraint within a conflict set) or the anomaly itself (e.g. delete all constraints within a diagnosis). Finally, the system also offers explanations for anomalies (see Section 4.2). For example, the system shows which constraints are responsible for a redundancy.

6 Interfaces for the Maintenance of Constraint-based Systems

For a detailed description of anomalies we refer the reader to the Sections 4.1 and 5.1 and [Reinfrank et al., 2015a; Felfernig et al., 2014e,d; Benavides et al., 2013; Felfernig et al., 2013a, 2012a, 2011b].

Dependency detection

In our iCone implementation we have one visualization for constraint dependencies and another one for variable dependencies.

First, **dependencies between constraints** shows the relationship between products, question, and constraints. Figure 6.10 shows the graphical representation of the content-based filtering algorithm (see Section 5.3).

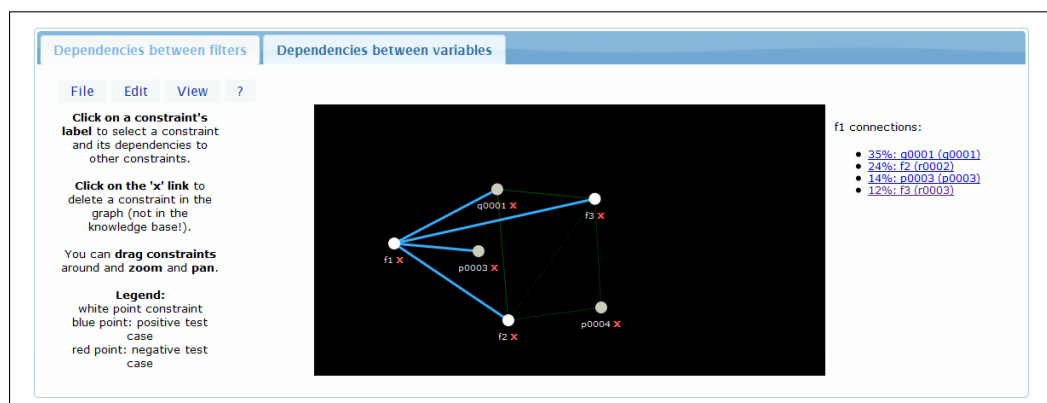


Figure 6.10: Dependencies of constraints. For example, the selected constraint c_1 has dependencies to the customer variable *usage_scenario*, constraints c_2 and c_3 and to the product $p0003$.

Figure 6.11 shows the **dependencies between variables**. These relationships can be calculated in two different ways depending on the checkbox 'Ignore all other variables'. If the checkbox is clicked, the system does exactly one consistency check for each possible combination of the domain elements for the variables selected in the drop down menus. The result is either o

6.2 iCone: an Interface for the Maintenance of Constraint-based Systems

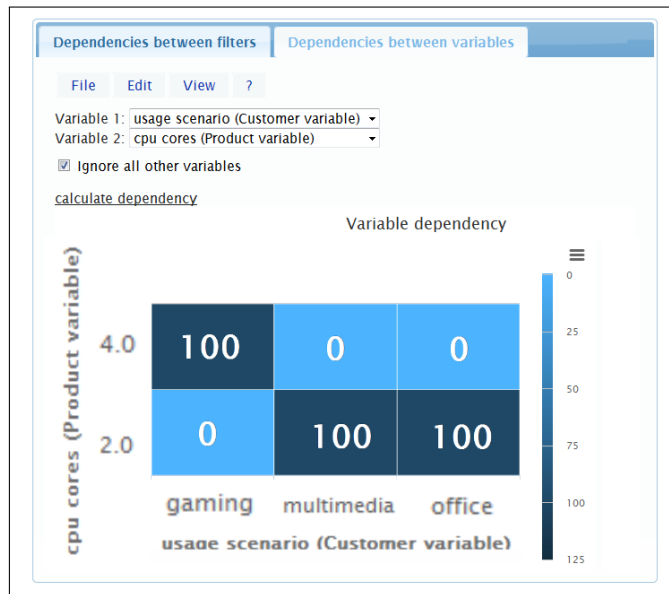


Figure 6.11: Dependencies of domain elements between two variables. For example, if variable $cpu_cores = 2$ and variable $usage_scenario = office$ then the probability that this combination is consistent is 100%.

or 100 for each combination of domain elements where 0 means that the combination of these two domain elements can never be consistent and 100 means that the combination is consistent when there are no other variable assignments.

If the user deselects the checkbox 'Ignore all other variables', the system approximates the probability that the combination of each domain element pair is consistent. The system generates constraints randomly, adds them to the set C_{KB} and checks if the knowledge base is consistent. For example, if the user selects the variables $usage_scenario$ and cpu_cores the combination $usage_scenario = office \wedge cpu_cores = 2$ is consistent but if the user disables the 'Ignore all other variables' checkbox, the system randomly adds constraints s.t. the initial configuration $usage_scenario = office \wedge cpu_cores = 2$ may be extended by $price = 599$ and the configuration would be inconsistent. The probability that the combination of domain elements is consistent, is displayed in Figure 6.11.

6 Interfaces for the Maintenance of Constraint-based Systems

While in simple knowledge bases it is possible to calculate all possible combinations of variable assignments, in medium and large knowledge bases with ten variables and each variable having five elements in its domain, we have approx. 10 million possible different combinations. While such sets of possible combinations can not be calculated in real-time systems, we use Gibbs sampling to approximate the number of consistent combinations. For example, if we have 1,000 checks and 300 are consistent, we expect that the number of consistent combinations is 30%. For a detailed description of the simulation strategy we refer the reader to [Reinfrank et al., 2015a,c] and Section 5.3.

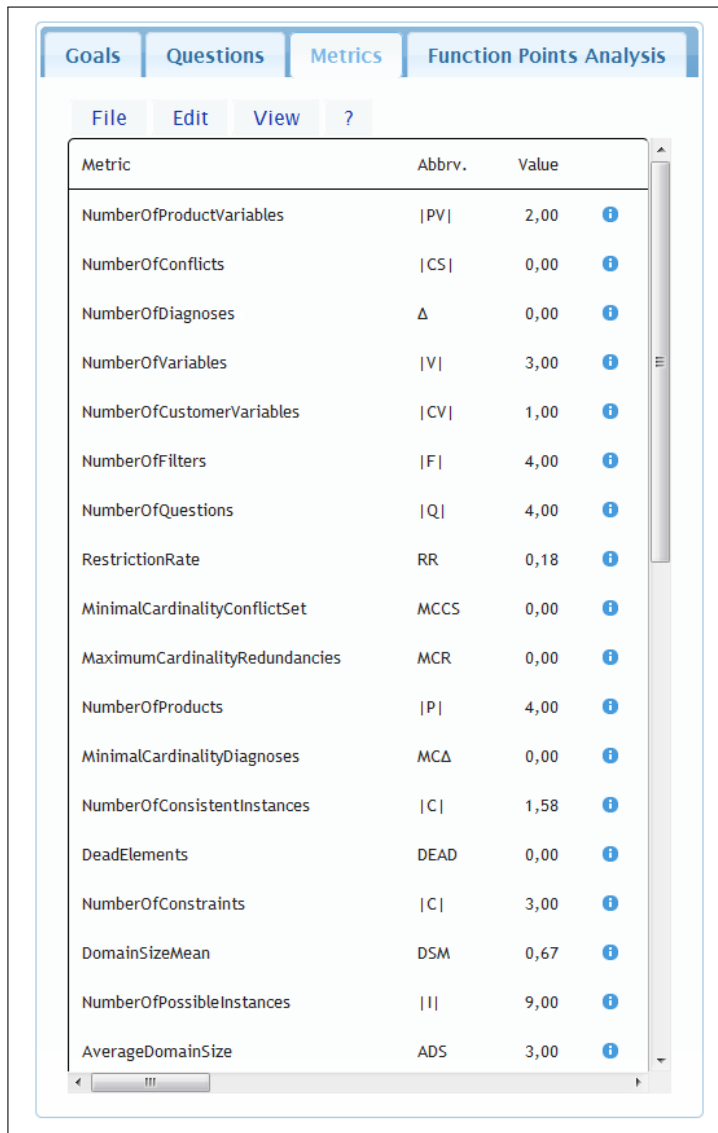
Metric calculation and evaluation

The iCone interface offers much information (products, questions, constraint, anomalies, dependencies, and test cases). To get an overview of the quality of the knowledge base, the iCone interface offers an overview of several metrics (see Figure 6.12).

While the number of metrics and their interpretation can be difficult, the interface supports knowledge engineers in the interpretation. First, the system shows changes in the values for each metric. Therefore the system picks up the latest versions of the knowledge base and shows the differences. Second, the iCone interface uses the metrics as a basis for the goal-question-metric (GQM) approach. Metrics will be aggregated for questions and they will be used to evaluate three goals. The goals **maintainability**, **understandability**, and **functionality** are generic goals for each knowledge base in the iCone system.

Finally, the system also uses the function point analysis (FPA) to analyze the quality of a knowledge base. The system uses the customer requirements as external input (EI), the available questions and products as external query (EQ), the number of offered products and configurations as external output (EO), items in the knowledge base (products, questions, constraints)

6.2 iCone: an Interface for the Maintenance of Constraint-based Systems



Metric	Abbrev.	Value	
NumberOfProductVariables	PV	2,00	i
NumberOfConflicts	CS	0,00	i
NumberOfDiagnoses	Δ	0,00	i
NumberOfVariables	V	3,00	i
NumberOfCustomerVariables	CV	1,00	i
NumberOfFilters	F	4,00	i
NumberOfQuestions	Q	4,00	i
RestrictionRate	RR	0,18	i
MinimalCardinalityConflictSet	MCCS	0,00	i
MaximumCardinalityRedundancies	MCR	0,00	i
NumberOfProducts	P	4,00	i
MinimalCardinalityDiagnoses	MCA	0,00	i
NumberOfConsistentInstances	C	1,58	i
DeadElements	DEAD	0,00	i
NumberOfConstraints	C	3,00	i
DomainSizeMean	DSM	0,67	i
NumberOfPossibleInstances	I	9,00	i
AverageDomainSize	ADS	3,00	i

Figure 6.12: Visualization of metrics. When the knowledge engineer clicks on the ‘i’ symbol, she will receive the history of the metric (see Figure 5.5). Each time when the knowledge base will be updated, a new version of the knowledge base is created and the knowledge engineer can see the history of this metric.

as internal logical files (ILF), and external information like the product assortment from an ERP-system as external interface file (EIF).

6 Interfaces for the Maintenance of Constraint-based Systems

For a detailed description of the metrics we refer the reader to Section 5.4 and [[Reinfrank et al., 2015a,b](#); [Bagheri and Gasevic, 2011](#); [Lethbridge, 1998](#)].

7 Conclusion

7.1 Summary

We have presented several techniques to support knowledge engineers in their maintenance tasks. First, we have given an overview of current techniques for the support in Chapter 2. Next, we described our understanding of constraint satisfaction problems (Section 3.1), listed some definitions in Section 3.2, and introduced running examples in Section 3.3.

Chapter 4 has taken a look into anomaly management. Section 4.1 expanded the term *anomaly management* - which currently could be used synonymously as *inconsistency management* (Section 4.1.1) - with the detection of redundancies (Section 4.1.2) and well-formedness violations (Section 4.1.3). The detection of those types of anomalies helps knowledge engineers to get a better understanding of the knowledge base. Section 4.1 has focused on the description of the anomalies. The term *management* subsumes different tasks related to anomalies. Therefore, in Section 4.2 we have shown how we can explain anomalies to knowledge engineers. While the previous sections expect that the knowledge base describes the real products, Section 4.3 has explained, how knowledge engineers can use test cases to detect and repair differences between the real product and the knowledge base. Finally, we have presented a study how the presentation of anomalies has an influence on the satisfaction and the time knowledge engineers need to repair an inconsistency (Section 4.4).

7 Conclusion

Chapter 5 has described several techniques that support knowledge engineers in their maintenance tasks. We have explained how collaborative filtering techniques can be used to differ between relevant and irrelevant information regarding a maintenance task (Section 5.1).

We have described algorithms to detect anomalies in Section 5.2. In addition to the current algorithms for calculating conflicts (Section 5.2.1), diagnoses (Section 5.2.1), and redundancies (Section 5.2.2) in a linear way, we have introduced algorithms to calculate redundancies with a divide-and-conquer algorithm (Section 5.2.2) and well-formedness violations (Section 5.2.3). We have also explained how we can calculate redundancies based on assignments instead of constraints to get a more detailed information about the anomaly than with calculating anomalies based on constraints in Section 5.2.2.

Simulation techniques are a well established technology in several application areas except in the domain of maintaining product configuration knowledge bases. We have taken this technology to detect dependencies of variables in Section 5.3. This technique helps knowledge engineers to understand how an assignment of a variable can influence the possible values (and the probability of the assignment of other values) of other variables.

Based on a literature review, we have discussed how we can evaluate a constraint-based product configuration knowledge base (Section 5.4). On the basis of the goal-question-metrics approach, we first described the goals a knowledge base has to fit with (Section 5.4.1). How we can relate the three goals with our questions, is described in Section 5.4.2. To evaluate the answers of the questions, a set of metrics has been identified in Section 5.4.3. Those metrics can have positive and negative implications to the questions. A description how to interpret the results for goals, questions, and metrics is difficult. Therefore, we have suggested to describe the changes of these values when a knowledge base is maintained (Section 5.4.4).

7.2 Further Research

In Section 4.3 we have illustrated how test cases can be used to correct a knowledge base. How we can generate the test cases automatically, has been explained in Section 5.5. We have introduced how we can generate test cases automatically using Gibbs' sampling, boundary values, and micro tasks.

The previously explained techniques to support knowledge engineers focus on the maintenance of product configuration knowledge bases. While these techniques help to work more efficiently, supporting the whole development process would lead to more effective maintenance. Considering that, we have taken a look into the development process in Section 5.6 and have given some initial hints how we can optimize the development process for product configuration knowledge bases.

While the previous Chapters 4 and 5 have concentrated on techniques to support knowledge engineers, Chapter 6 has focused on user interfaces for configuration knowledge bases. Section 6.1 has listed requirements for such interfaces and presented some examples from practice how the requirements can be realized. Our implemented system iCone is presented in Section 6.2. This system shows how we have realized all of the previously explained techniques to support knowledge engineers and their maintenance tasks.

7.2 Further Research

We have presented several techniques that help to support knowledge engineers in their knowledge base maintenance tasks. These techniques can be used in almost each product domain. We assume that these techniques tackle the main issues of maintenance. Qualitative and quantitative studies have to be done to optimize the supporting techniques with *focus on product domains*.

7 Conclusion

In Section 5.2.2 *assignment-based redundancy detection* a) detects more redundancies than constraint-based redundancy detection and b) gives a more detailed information about the anomaly. As recommended in Section 5.2.2, this technology can also be applied for inconsistency detection. For example, an assignment based inconsistency detection for the constraints $c_1 = x < 5 \wedge y = 3; c_2 = x > 5$ returns $CS_1 = \{\{x < 5\} \in c_1, \{x > 5\} \in c_2\}$ instead of $CS_1 = \{c_1, c_2\}$ and gives a more detailed description for the inconsistency. The diagnoses for this inconsistency would be more detailed, too, because $\Delta_1 = \{\{x < 5\} \in c_1\}$ and $\Delta_2 = \{\{x > 5\} \in c_2\}$ fits more to the original knowledge base.

As explained in Section 5.6, we can increase the effectiveness of a knowledge base maintenance task. A lot of research related to the *development process* of knowledge bases has been in done in many research areas similar to constraint-based product configuration systems. In this thesis we have given a first insight into a development process for product configuration systems but a lot of research should be done to get a deep understanding of the development process.

Bibliography

- Abreu, F. B. and Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. *Proceedings of the 3rd international software metrics symposium*, pages 90 – 99.
- Anderson, D. (1997). *Agile Product Development for Mass Customization*. McGraw-Hill.
- Angele, J., Fensel, D., Landes, D., and Studer, R. (1998). Developing knowledge-based systems with mike. *Automated Software Engineering*, 5(4):389–418.
- Bagheri, E. and Gasevic, D. (2011). Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3):579–612.
- Barker, V., O’Conner, D., Bachant, J., and Soloway, E. (1989). Expert systems for configuration at digital: Xcon and beyond. *Communications of the ACM*, 32(3):298 – 318.
- Barr, V. (1997). Applications of rule-base coverage measures to expert system evaluation. *AAAI*.
- Baumeister, J., Puppe, F., and Seipel, D. (2004). Refactoring methods for knowledge bases. In *Engineering Knowledge in the age of the Semantic Web: 14th international conference, EKAW, LNAI 3257*, pages 157–171. Springer.
- Benavides, D., Felfernig, A., Galindo, J. A., and Reinfrank, F. (2013). Automated analysis in feature modelling and product configuration. *ICSR*, pages 160 – 175.

Bibliography

- Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35:615–636.
- Bharadwaj, N., Naylor, R. W., and ter Hofstede, F. (2009). Consumer response to and choice of customized versus standardized systems. *International Journal of Research in Marketing*, 26(3):216 – 227.
- Blecker, T., Abdelkafi, N., Kreutler, G., and Friedrich, G. (2004). Product configuration systems: State of the art, conceptualization and extensions. *Eight Maghrebian Conference on Software Engineering and Artificial Intelligence*, pages 25 – 36.
- Blythe, J., Kim, J., Ramachandran, S., and Gil, Y. (2001a). An integrated environment for knowledge acquisition. *IUI*, pages 14 – 17.
- Blythe, J., Kim, J., Ramachandran, S., and Gil, Y. (2001b). An integrated environment for knowledge acquisition. In *Proceedings of the 6th international conference on Intelligent user interfaces, IUI '01*, pages 13–20, New York, NY, USA. ACM.
- Boehm, B. (1984). Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88.
- Bourke, R. (2000). Product configurators: Key enabler for mass customization - an overview. <http://www.pdmic.com/articles/midrange/-Aug2000.html> (Retrieval 10. Nov. 2003).
- Bowen, J. and Bahler, D. (1991). Conditional existence of variables in generalised constraint networks. In *AAAI*, pages 215–220. Citeseer.
- Briand, L., Wust, J., Ikonomovski, S., and Lounis, H. (1999). Investigating quality factors in object-oriented designs: an industrial case study. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 345–354.
- Brooke, J. (1986). Sus: a quick and dirty usability scale. In Jordan, P., Thomas, B., Weerdmeester, B., and McClelland, I. L., editors, *Usability Evaluation in Industry*. Taylor and Francis.

- Burke, R. (2000). Knowledge-based recommender systems. In *Encyclopedia of library and information systems*, page 2000. Marcel Dekker.
- Burke, R., Felfernig, A., and Goeker, M. (2011). The Future of Recommender Systems: Research and Applications. *AI Magazine*, 32(3):13–18.
- Burke, R. D., Hammond, K. J., and Yound, B. C. (1997). The findme approach to assisted browsing. *IEEE Expert*, 12(4):32–40.
- Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Comput. Surv.*, 41:15:1–15:58.
- Chen, L. and Pu, P. (2006). Evaluating critiquing-based recommender agents. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1, AAAI'06*, pages 157–162. AAAI Press.
- Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476 – 493.
- Chklovski, T. and Gil, Y. (2005). An analysis of knowledge collected from volunteer contributors. In Veloso, M. M. and Kambhampati, S., editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 564–571. AAAI Press / The MIT Press.
- Claessen, K., Een, N., Sheeran, M., and Sorensson, N. (2008). Sat-solving in practice. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 61–67.
- Daaboul, J., Bernard, A., and Laroche, F. (2009). Implementing Mass Customization: Literature Review. In *proceedings of the 5th World Conference on Mass Customization and Personalization*, pages –, Helsinki, Finland.
- Domshlak, C., Hüllermeier, E., Kaci, S., and Prade, H. (2011). Preferences in ai: An overview. *Artificial Intelligence*, 175(7–8):1037 – 1052. Representing, Processing, and Learning Preferences: Theoretical and Practical Challenges.

Bibliography

- Fahad, M. and Qadir, M. A. (2008). A framework for ontology evaluation. In Eklund, P. W. and Haemmerlé, O., editors, *Supplementary Proceedings of the 16th International Conference on Conceptual Structures, ICCS 2008, Toulouse, France, July 7-11, 2008*, volume 354 of *CEUR Workshop Proceedings*, pages 149–158. CEUR-WS.org.
- Falkner, A., Felfernig, A., and Haag, A. (2011). Recommendation Technologies for Configurable Products. *AI Magazine*, 32(3):99–108.
- Falkner, A. and Schreiner, H. (2014). *SIEMENS: Configuration and Reconfiguration in Industry*, pages 199 – 210. Volume 1 of [Felfernig et al. \[2014c\]](#).
- Felfernig, A. (2004). Effort estimation for knowledge-based configuration systems. In Maurer, F. and Ruhe, G., editors, *SEKE*, pages 148–154.
- Felfernig, A. (2007). Standardized configuration knowledge representations as technological foundation for mass customization. *IEEE Transactions on Engineering Management*, 54:41–56.
- Felfernig, A., Benavides, D., Galindo, J. A., and Reinfrank, F. (2013a). Towards anomaly explanation in feature models. *Workshop on Configuration*, pages 117 – 124.
- Felfernig, A., Blazek, P., Reinfrank, F., and Ninaus, G. (2014a). *Intelligent User Interfaces for Configuration Environments*, pages 89 – 106. Volume 1 of [Felfernig et al. \[2014c\]](#).
- Felfernig, A. and Burke, R. (2008). Constraint-based recommender systems: technologies and research issues. In *Proceedings of the 10th international conference on Electronic commerce, ICEC '08*, pages 3:1–3:10, New York, NY, USA. ACM.
- Felfernig, A., Friedrich, G., Gula, B., Hitz, M., Kruggel, T., Leitner, G., Melcher, R., Riepan, D., Strauß, S., Teppan, E., and Vitouch, O. (2007a). Persuasive recommendation: Serial position effects in knowledge-based recommender systems. In Kort, Y., IJsselsteijn, W., Midden, C., Eggen, B., and Fogg, B., editors, *Persuasive Technology*, volume 4744 of *Lecture Notes in Computer Science*, pages 283–294. Springer Berlin Heidelberg.

- Felfernig, A., Friedrich, G., Jannach, D., and Stumptner, M. (2004a). Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213 – 234.
- Felfernig, A., Friedrich, G., Jannach, D., and Stumptner, M. (2004b). Consistency-based Diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213–234.
- Felfernig, A., Friedrich, G., Jannach, D., and Zanker, M. (2006a). An integrated environment for the development of knowledge-based recommender applications. *Int. J. Electron. Commerce*, 11(2):11–34.
- Felfernig, A., Friedrich, G., Schubert, M., Mandl, M., Mairitsch, M., and Teppan, E. (2009). Plausible repairs for inconsistent requirements. *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 791–796.
- Felfernig, A., Friedrich, G. E., and Jannach, D. (2000). Uml as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 10:449–469.
- Felfernig, A., Gula, B., Leitner, G., Maier, M., Melcher, R., and Teppan, E. (2008a). Persuasion in knowledge-based recommendation. In Oinas-Kukkonen, H., Hasle, P., Harjumaa, M., Segerstahl, K., and Øhrstrøm, P., editors, *Persuasive Technology: Third International Conference, PERSUASIVE 2008, Oulu, Finland, June 4-6, 2008.*, pages 71–82, Berlin, Heidelberg. Springer.
- Felfernig, A., Gula, B., and Teppan, E. (2006b). Knowledge-based Recommender Technologies for Marketing and Sales. *Special issue of Personalization Techniques for Recommender Systems and Intelligent User Interfaces for the International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 21(2):1–22.
- Felfernig, A., Haas, S., Ninaus, G., Schwarz, M., Ulz, T., and Stettinger, M. (2014b). Recturk: Constraint-based recommendation based on human computation. *CrowdRec*.

Bibliography

- Felfernig, A., Hotz, L., Bagley, C., and Tiihonen, J., editors (2014c). *Knowledge-based configuration. From research to business cases*, volume 1. Morgan Kaufmann.
- Felfernig, A., Isak, K., and Russ, C. (2006c). Knowledge-based recommendation: Technologies and experiences from projects. In *Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy*, pages 632–636, Amsterdam, The Netherlands, The Netherlands. IOS Press.
- Felfernig, A., Isak, K., Szabo, K., and Zachar, P. (2007b). The vita financial services sales support environment. In *Proceedings of the 19th National Conference on Innovative Applications of Artificial Intelligence - Volume 2, IAAI'07*, pages 1692–1699. AAAI Press.
- Felfernig, A., Mandl, M., Pum, A., and Schubert, M. (2010). Empirical knowledge engineering: Cognitive aspects in the development of constraint-based recommenders. In García-Pedrajas, N., Herrera, F., Fyfe, C., Benítez, J. M., and Ali, M., editors, *Trends in Applied Intelligent Systems*, volume 6096 of *Lecture Notes in Computer Science*, pages 631–640. Springer Berlin / Heidelberg.
- Felfernig, A., Reinfrank, F., and Ninaus, G. (2012a). Resolving anomalies in configuration knowledge bases. *ISMIS*, 1(1):1 – 10.
- Felfernig, A., Reinfrank, F., Ninaus, G., and Blazek, P. (2014d). *Redundancy Detection in Configuration Knowledge*, pages 157 – 166. Volume 1 of [Felfernig et al. \[2014c\]](#).
- Felfernig, A., Reiterer, S., Reinfrank, F., Ninaus, G., and Jeran, M. (2014e). *Conflict Detection and Diagnosis in Configuration*, pages 73 – 87. Volume 1 of [Felfernig et al. \[2014c\]](#).
- Felfernig, A., Reiterer, S., Stettinger, M., Reinfrank, F., Jeran, M., and Ninaus, G. (2013b). Recommender systems for configuration knowledge engineering. *Workshop on Configuration*, pages 51 – 54.
- Felfernig, A., Schippel, S., Leitner, G., Reinfrank, F., Isak, K., Mandl, M., Blazek, P., and Ninaus, G. (2013c). Automated repair of scoring rules in constraint-based recommender systems. *AI Commun.*, 26(1):15–27.

- Felfernig, A. and Schubert, M. (2011a). Personalized Diagnoses for Inconsistent User Requirements. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing (AIEDAM)*, 25(1):115–129.
- Felfernig, A. and Schubert, M. (2011b). Personalized diagnoses for inconsistent user requirements. *AI EDAM*, 25(2):175–183.
- Felfernig, A., Schubert, M., and Reiterer, S. (2013d). Personalized diagnosis for over-constrained problems. *IJCAI*, pages 1990 – 1996.
- Felfernig, A., Schubert, M., and Zehentner, C. (2011a). An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing (AIEDAM)*, 25(2):175–184.
- Felfernig, A., Schubert, M., and Zehentner, C. (2012b). An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62.
- Felfernig, A. and Shchekotykhin, K. (2006). Debugging user interface descriptions of knowledge-based recommender applications. In *Proceedings of the 11th International Conference on Intelligent User Interfaces, IUI '06*, pages 234–241, New York, NY, USA. ACM.
- Felfernig, A., Teppan, E., Friedrich, G., and Isak, K. (2008b). Intelligent debugging and repair of utility constraint sets in knowledge-based recommender applications. In *Proceedings of the 13th International Conference on Intelligent User Interfaces, IUI '08*, pages 217–226, New York, NY, USA. ACM.
- Felfernig, A., Zehentner, C., and Blazek, P. (2011b). Corediag: Eliminating redundancy in constraint sets. In Sachenbacher, M., Dressler, O., and Hofbaur, M., editors, *DX 2011. 22nd International Workshop on Principles of Diagnosis*, pages 219 – 224, Murnau, GER.
- Felfernig, A., Zehentner, C., and Blazek, P. (2011c). Corediag: Eliminating redundancy in constraint sets. *22nd International Workshop on Principles of Diagnosis (DX'11)*, pages 219–224.
- Fisher, M. L. and Ittner, C. D. (1999). The impact of product variety on automobile assembly operations: Empirical evidence and simulation analysis. *Management Science*, 45:771–786.

Bibliography

- Fleischanderl, G., Friedrich, G. E., Haselböck, A., Schreiner, H., and Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68.
- Fogliatto, F. S., da Silveira, G. J., and Borenstein, D. (2012). The mass customization decade: An updated review of the literature. *International Journal of Production Economics*, 138(1):14 – 25.
- Fourer, R., Gay, D. M., and Kernighan, B. W. (2002). *AMPL: A modeling language for mathematical programming*. Cole Publishing Company, 2 edition.
- Freuder, E. (1997). In pursuit of the holy grail. *Constraints*, 2(1):57–61.
- Friedrich, G., Jannach, D., Stumptner, M., and Zanker, M. (2014). *Knowledge Engineering for configuration systems*, pages 139 – 155. Volume 1 of [Felfernig et al. \[2014c\]](#).
- Friedrich, G. and Zanker, M. (2011). A Taxonomy for Generating Explanations in Recommender Systems. *AI Magazine*, 32(3):90–98.
- Gartner (2005). Top ten risks to a configuration project and how to avoid them.
- Gershberg, F. B. and Shimamura, A. P. (1994). Serial position effects in implicit and explicit tests of memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 20:1370–1378.
- Grabner-Kraeuter, S. and Kaluscha, E. (2003). Empirical research in on-line trust: a review and critical assessment. *International Journal of Human-Computer Studies*, 58(6):783–812.
- Grimm, S. and Wissmann, J. (2011). Elimination of redundancy in ontologies. In Antoniou, G., Grobelnik, M., Simperl, E. P. B., Parsia, B., Plexousakis, D., Leenheer, P. D., and Pan, J. Z., editors, *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*, volume 6643 of *Lecture Notes in Computer Science*, pages 260–274. Springer.
- Günter, A. and Kühn, C. (1999). Knowledge-Based Configuration - Survey and Future Directions. In Puppe, F., editor, *XPS-99: Knowledge Based*

- Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, Springer Lecture Notes in Artificial Intelligence 1570, Würzburg.
- Hartmann, H. and Trew, T. (2008). Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Proceedings of the 2008 12th International Software Product Line Conference*, pages 12–21, Washington, DC, USA. IEEE Computer Society.
- Hong, J.-C. and Liu, M.-C. (2003). A study on thinking strategy between experts and novices of computer games. *Computers in Human Behavior*, 19:245 – 258.
- Hotz, L., Felfernig, A., Stumptner, M., Ryabokon, A., Bagley, C., and Wolter, K. (2014). *Configuration Knowledge Representation and Reasoning*, pages 41 – 72. Volume 1 of [Felfernig et al. \[2014c\]](#).
- Hu, S., Zhu, X., Wang, H., and Koren, Y. (2008). Product variety and manufacturing complexity in assembly systems and supply chains. *CIRP Annals - Manufacturing Technology*, 57(1):45 – 48.
- Jannach, D., Zanker, M., Felfernig, A., and Friedrich, G. (2010). *Recommender Systems: An Introduction*, volume 1. University Press, Cambridge.
- Jeffress, L. A., editor (1951). *Cerebral Mechanisms in Behaviour: The Hixon Symposium*. Hafner Publishing Co Ltd.
- John, U. and Geske, U. (2003). Web knowledge management and decision support: 14th international conference on applications of prolog, inap 2001 tokyo, japan, october 20–22, 2001 revised papers. In Bartenstein, O., Geske, U., Hannebauer, M., and Yoshie, O., editors, *Web Knowledge Management and Decision Support: 14th International Conference on Applications of Prolog, INAP 2001 Tokyo, Japan, October 20–22, 2001 Revised Papers*, chapter Constraint-Based Configuration of Large Systems, pages 217–232. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Juengst, W. E. and Heinrich, M. (1998). Using resource balancing to configure modular systems. *IEEE Intelligent Systems*, 13(4):50–58.

Bibliography

- Junker, U. (2004). Quickxplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artificial intelligence, AAAI'04*, pages 167–172. AAAI Press.
- Kahneman, D., Knetsch, J., and Thaler, R. H. (1991). Anomalies: The endowment effect, loss aversion, and status quo bias. *The Journal of Economic Perspectives*, 5:193 – 206.
- Keefe, R. M. O. and Preece, A. D. (1996). The development, validation and implementation of knowledge-based systems. *European Journal of Operational Research*, 92(3):458 – 473.
- Keeney, R. and Raiffa, H. (1993). *Decisions with Multiple Objectives: Preferences and Value Trade-Offs*. Wiley series in probability and mathematical statistics. Applied probability and statistics. Cambridge University Press.
- Keeney, R. L. and Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley and Sons.
- Kellner, M. I., Madachy, R. J., and Raffo, D. M. (1999). Software process simulation modeling: Why? what? how? *Journal of Systems and Software*, 46(2–3):91 – 105.
- Konstan, J., Miller, B., Maltz, D., Herlocker, J., Gordon, L., and Riedl, J. (1997). GroupLens: applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87.
- Lashley, K. (1951). The problem of serial order in behavior. In [Jeffress \[1951\]](#), page 506–528.
- Lauenroth, K. and Pohl, K. (2007). Towards automated consistency checks of product line requirements specifications. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07*, pages 373–376, New York, NY, USA. ACM.
- Lethbridge, T. (1998). Metrics for concept-oriented knowledge bases. *International Journal of Software Engineering and Knowledge Engineering*, 8:16–1.
- Mandl, M., Felfernig, A., Teppan, E., and Schubert, M. (2011a). Consumer decision making in knowledge-based recommendation. *Journal of Intelligent Information Systems*, 37(1):1–22.

- Mandl, M., Felfernig, A., Tiihonen, J., and Isak, K. (2011b). Status quo bias in configuration systems. In Mehrotra, K. G., Mohan, C. K., Oh, J. C., Varshney, P. K., and Ali, M., editors, *IEA/AIE (1)*, volume 6703 of *Lecture Notes in Computer Science*, pages 105–114. Springer.
- McSherry, D. (2003). Similarity and compromise. In *Proceedings of the Fifth International Conference on Case-Based Reasoning*, pages 291–305. Springer.
- McSherry, D. (2004). Maximally successful relaxations of unsuccessful queries. In *15th Irish Conference on Artificial Intelligence and Cognitive Science*, AICS-04, pages 127–136.
- Mehrotra, M., Bobrovnikoff, D., Chaudhri, V., and Hayes, P. (2002). A clustering approach for knowledge base analysis. *American Association for Artificial Intelligence*.
- Mirzadeh, N., Ricci, F., and Bansal, M. (2005). Feature selection methods for conversational recommender systems. In *2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, pages 772–777.
- Musen, M. A., Eriksson, H., Gennari, J. H., and Tu, S. W. a. (1994). Protégé-ii: A suite of tools for development of intelligent systems from reusable components. *Proc Annu Symp Comput Appl Med Care*.
- Myers, G. J., Badgett, T., and Sandler, C. (2012). *The art of software testing*. John Wiley & Sons, 3 edition.
- Nabil, D., El-Korany, A., and Eldin, A. S. (2008). Towards a suite of quality metrics for kadss-domain knowledge. *Expert Systems with Applications*, 35:654 – 660.
- Neumaier, A., Shcherbina, O., Huyer, W., and Vinkó, T. (2005). A comparison of complete global optimization solvers. *Mathematical Programming*, 103(2):335–356.
- O’Keefe, R. M. and O’Leary, D. E. (1993). A review and survey of expert system verification and validation. *Artificial Intelligence Review*, 7(1):3 – 42.
- Olhager, J. and Persson, F., editors (2007). *Advances in Production Management Systems*. Springer US, Linköping, Sweden, 1 edition.

Bibliography

- Pazzani, M. and Billsus, D. (1997). Learning and revising user profiles: The identification of interesting web sites. *Machine Learning*, 27(3):313–331.
- Pazzani, M. and Billsus, D. (2007). Content-based recommendation systems. In Brusilovsky, P., Kobsa, A., and Nejdl, W., editors, *The Adaptive Web*, volume 4321 of *Lecture Notes in Computer Science*, pages 325–341. Springer Berlin / Heidelberg.
- Piette, C. (2008). Let the solver deal with redundancy. In *Proceedings of the 2008 20th IEEE International Conference on Tools with Artificial Intelligence - Volume 01*, pages 67–73, Washington, DC, USA. IEEE Computer Society.
- Pine, B. and Davis, S. (1999). *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press.
- Pine, J. B., editor (1992). *Mass Customization. The new frontier in business competition*. Harvard Business School.
- Preece, A. (1998). Building the right system right evaluating v&v methods in knowledge engineering.
- Preece, A. D. and Shinghal, R. (1994). Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, 9(8):683–701.
- Preece, A. D., Shinghal, R., and Batarekh, A. (1992). Principles and practice in verifying rule-based systems. *The Knowledge Engineering Review*, 7:115–141.
- Preece, A. D., Talbot, S., and Vignollet, L. (1997). Evaluation of verification tools for knowledge-based systems. *International Journal of Human-Computer Studies*, 47(5):629 – 658.
- Prerau, D. S. (1987). Knowledge acquisition in the development of a large expert system. *AI Magazine*, 8(2):43–51.
- Rabe, M. and Jäkel, F.-W. (2002). Decomposition of large simulation systems for supply chains and manufacturing systems.

- Rabiser, R. and Dhungana, D. (2007). Integrated support for product configuration and requirements engineering in product derivation. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 219–228.
- Randall, T., Terwiesch, C., and Ulrich, K. (2005). Principles for User Design of Customized Products. *California Management Review*, 47(4):1–18.
- Reichwald, R. and Piller, F. (2009). *Interaktive Wertschöpfung. Open Innovation, Individualisierung und neue Formen der Arbeitsteilung*. Gabler, Wiesbaden, 2 edition.
- Reinfrank, F., Ninaus, G., and Felfernig, A. (2015a). Intelligent techniques for the maintenance of constraint-based systems. *Configuration Workshop*.
- Reinfrank, F., Ninaus, G., Peischl, B., and Wotawa, F. (2015b). A goal-question-metrics model for configuration knowledge bases. *Configuration Workshop*.
- Reinfrank, F., Ninaus, G., Wotawa, F., and Felfernig, A. (2015c). Maintaining constraint-based configuration systems: Challenges ahead. *Configuration Workshop*.
- Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95.
- Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994). GroupLens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work, CSCW '94*, pages 175–186, New York, NY, USA. ACM.
- Ricci, F. and Nguyen, Q. N. (2007). Acquiring and revising preferences in a critique-based mobile recommender system. *IEEE Intelligent Systems*, 22(3):22–29.
- Richardson, M. and Domingos, P. M. (2003). Building large knowledge bases by mass collaboration. In Gennari, J. H., Porter, B. W., and Gil, Y., editors, *Proceedings of the 2nd International Conference on Knowledge Capture (K-CAP 2003), October 23-25, 2003, Sanibel Island, FL, USA*, pages 129–137. ACM.

Bibliography

- Rogoll, T. and Piller, F. (2004). Product configuration from the customer's perspective: a comparison of configuration systems in the apparel industry.
- Sabin, D. and Weigel, R. (1998). Product Configuration Frameworks – A Survey. *IEEE Intelligent Systems*, 13(4):42–49.
- Sabin, M. and Freuder, E. C. (1998). Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. *Proceeding of Constraint Programming (CP'98)*.
- Salvetto, P. F., Martinez, M. F., Luna, C. D., and Segovia, J. (2004). A very early estimation of software development time and effort using neural networks. Workshop de Ingeniería de Software y Base de Datos.
- Samuelson, W. and Zeckhauser, R. (1988). Status quo bias in decision making. *Journal of Risk and Uncertainty*, 1:7–59.
- Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web, WWW '01*, pages 285–295, New York, NY, USA. ACM.
- Schmitt, C., Dengler, D., and Bauer, M. (2003). Multivariate preference models and decision making with the maut machine. In Brusilovsky, P., Corbett, A., and de Rosis, F., editors, *User Modeling 2003*, volume 2702 of *Lecture Notes in Computer Science*, pages 297–302. Springer Berlin Heidelberg.
- Schreiber, G., Wielinga, B., de Hoog, R., Akkermans, H., and Van de Velde, W. (1994). Commonkads: a comprehensive methodology for kbs development. *IEEE Expert*, 9(6):28–37.
- Schubert, M., Felfernig, A., and Reinfrank, F. (2011). Reaction: Personalized minimal repair adaptations for customer requests. In Christiansen, H., Tré, G. D., Yazici, A., Zadrozny, S., Andreasen, T., and Larsen, H. L., editors, *Flexible Query Answering Systems - 9th International Conference, FQAS 2011, Ghent, Belgium, October 26-28, 2011 Proceedings*, volume 7022 of *Lecture Notes in Computer Science*, pages 13–24. Springer.

- Shortliffe, E. H. and Patel, V. L. (2007). 9 - human-intensive techniques. In Greenes, R. A., M.D., and Ph.D., editors, *Clinical Decision Support*, pages 207 – 226. Academic Press, Burlington.
- Simonson, I. (2003). Determinants of Customer's Responses to Customized Offers: Conceptual Framework and Research Propositions. *Stanford GSB Working Paper No. 1794*.
- Smyth, B., McGinty, L., Reilly, J., and McCarthy, K. (2004). Compound critiques for conversational recommender systems. In *Web Intelligence, 2004. WI 2004. Proceedings. IEEE/WIC/ACM International Conference on*, pages 145–151.
- Studer, R., Benjamins, V. R., and Fensel, D. (1998). Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25:161 – 197.
- Stumptner, M., Friedrich, G., and Haselböck, A. (1998). Generative Constraint-based Configuration of Large Technical Systems. *AI EDAM*, 12(04):307–320.
- Thompson, C. A., Göker, M. H., and Langley, P. (2004). A personalized system for conversational recommendations. *J. Artif. Int. Res.*, 21(1):393–428.
- Tiihonen, J. and Felfernig, A. (2010). Towards Recommending Configurable Offerings. *International Journal of Mass Customization*, 3(4):389–406.
- Tiihonen, J. and Soinen, T. (1997). *Product Configurators: Information System Support for Configurable Products*. Helsinki University of Technology.
- Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press.
- Tversky, A. and Kahneman, D. (1974). Judgement under uncertainty: Heuristics and biases. *Science*, 185(4157):1124 – 1131.
- Tversky, A., Slovic, P., and Kahneman, D. (1990). The causes of preference reversal. *American Economic Review*, 80(1):204–17.
- van Melle, W., Shortliffe, E. H., and Buchanan, B. G. (1984). Emycin: A knowledge engineer's tool for constructing rule-based expert systems. In

Bibliography

- Buchanan, B. G. and Shortliffe, E. H., editors, *Rule-Based Expert Systems. The Mycin Experiments of the Stanford Heuristic Programming Project*, pages 302–313. Addison-Wesley.
- von Winterfeldt, D. and Edwards, W. (1986). *Decision Analysis and Behavioral Research*. Cambridge University Press.
- Winterfeldt, D. and Edwards, W. (1986). *Decision Analysis and Behavioral Research*. Cambridge University Press.
- Wotawa, F., Reinfrank, F., Ninaus, G., and Felfernig, A. (2015a). icone: intelligent environment for the development and maintenance of configuration knowledge bases. *IJCAI 2015 Joint Workshop on Constraints and Preferences for Configuration and Recommendation*.
- Wotawa, F., Stettinger, M., Reinfrank, F., Ninaus, G., and Felfernig, A. (2015b). Conflict management for constraint-based recommendation. *IJCAI 2015 Joint Workshop on Constraints and Preferences for Configuration and Recommendation*.
- Zwicky, F. (1966). *Entdecken, Erfinden, Forschen im morphologischen Weltbild*. Droemer-Knauer.