Dissertation

Consistency Management in Constraint-Based Systems

Monika Schubert

Graz, 2011

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. DI. Dr. Alexander Felfernig
Second reviewer: Univ.-Prof. DI. Dr. Gerhard Friedrich

Abstract (English)

Constraint-based applications can efficiently support customers in finding suitable items (products or services). The main applications areas of constraint programming comprise scheduling and planning as well as recommendation and configuration. Most of these applications are interactive ones, meaning that customers explicitly specify their requirements or needs. Based on these requirements or needs the system tries to find relevant items. If the system cannot determine any item that satisfies all specified requirements, then an inconsistent situation occurred. In such a situation, intelligent consistency management techniques are needed to support the customer in finding a way to a consistent state. State-of-the-art approaches focus mainly on minimal diagnoses to show a minimal set of faulty requirements that need to be adapted in order to restore consistency. Although the available approaches help the customers, they show big deficits in the integration of different personalization methods. Additionally, the hard real-time requirement of interactive constraint-based systems is often not addressed. For this reason, even small-sized problems soon become computationally intractable.

This thesis introduces different approaches and techniques to support customers in restoring consistency when interacting with constraint-based systems, especially with recommender and configuration systems. In order to solve the challenge of personalizing the presented set of faulty requirements, a similarity-based, a utility-based, a probability-based as well as a hybrid approach are presented. Empirical studies clearly show the improvements in terms of prediction quality of the developed personalization approaches. Additionally, this thesis presents approaches which help to significantly improve the runtime of consistency management. Finally, algorithms are introduced which are addressing both challenges, namely the improvements of the prediction quality as well as run time improvements.

Zusammenfassung / Abstract (German)

Constraint-basierte Systeme bieten Kunden eine effiziente Hilfe bei der Suche nach geeigneten Produkten und Dienstleistungen. Die wichtigsten Anwendungsgebiete sind Scheduling, Planung, Empfehlung und Konfiguration. Die meisten dieser Anwendungen haben eine interaktive Benutzeroberfläche, in der die Kunden ihre Anforderungen und Bedürfnisse eingeben können. Das System versucht ein Produkt oder eine Dienstleistung zu finden, welches diesen Anforderungen und Bedürfnissen entspricht. Sollte es nicht möglich sein, ein Produkt zu finden, das den gestellten Anforderungen genügt, dann entsteht eine Inkonsistenz zwischen den Kundenanforderungen und dem Produktsortiment. Um dem Kunden bei der Produktsuche zu unterstützen, können Verfahren und Techniken der Künstlichen Intelligenz eingesetzt werden. Ein verbreiteter Ansatz zum Umgang mit den erwähnten Inkonsistenzen ist die Berechnung von minimalen Diagnosen, d.h., einer minimalen Menge von fehlerhaften Anforderungen, die vom Kunden zur Beseitigung der Inkonsistenz angepaßt werden müssen. Obwohl diese Methode den Kunden hilft, gibt es dennoch große Defizite im Bereich der Integration von Personalisierungskonzepten. Darüber hinaus gibt es harte Laufzeitanforderungen von interaktiven Anwendungen, die von konventionellen Berechnungsverfahren selbst bei einfachen Diagnoseaufgaben nicht erfüllt werden können.

In dieser Arbeit werden verschiedene Ansätze dargestellt, die dazu verwendet werden können, Inkonsistenzen aufzulösen. Zur Verbesserung der Verwendbarkeit existierender Diagnoseansätze, werden unterschiedliche Ansätze zur Individualisierung (unter anderem basierend Ähnlichkeitswerten, Wahrscheinlichkeitswerten oder Nutzwerten) präsentiert. Weiters wird ein hybrider Ansatz vorgestellt, welcher die individuellen Personalisierungsansätze kombiniert. Mit Hilfe von empirischen Studien werden die Ansätze bezüglich ihrer Vorhersagequalität evaluiert. Darber hinaus werden Ansätze präsentiert, die eine Laufzeitverbesserung bringen. Letztendlich werden Algorithmen eingefhrt, die sich mit beiden Herausforderungen befassen, nämlich der Verbesserung der Vorhersagequalität und der Laufzeitverbesserung.

Alexander Felfernig, Arabella Gaß, Birgit Hofer,
Burghild Schubert, Christian Körner, Christine Antensteiner,
Christoph Bauer, Christoph Zehentner, **Colleagues**, Daniel Muschick, Elfriede Schubert,
Elisabeth Jöbstl, Erich Teppan, **Family**, Francesco Ricci, Franz Wotawa, **Friends**, Georg Piewald,
Gerald Steinbauer, Gerhard Friedrich, Gerhard Leitner, Harald Kröll, Holger Klier, **Institute for Software Technology**, Isabella Plimon, Juha Tiihonen, Karl Voit, Mark Kröll, Markus Mairitsch,
Markus Strohmaier, Martina Koitz, Michael Reip, Monika Mandl, Oswin Aichholzer,
Peter Prettenhofer, Peter Schubert, Petra Pichler, Philipp Ghirardini,
Ralph Koitz, Raphael Deimel, Stefan Galler, Stefan Schippl,
Stela Bulic-Wehrschütz, Stephan Gspandl, Thomas Schubert,
TUGraz, Verena Gangl, Walid Maalej, Werner Antensteiner,
Wolfgang Slany, *...thank you*

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, ______________________

Place, Date

____

Signature

Eidesstattliche Erklärung

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 entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ______________________

Ort, Datum

____

Unterschrift

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xv

1. Introduction 1

1.1. Motivation . . . 1
1.2. Research Objectives . . . 5
1.3. Contributions . . . 6
1.3.1. Contributions to the Field of Recommender Systems . . . 7
1.3.2. Contributions to the Field of Configuration Systems . . . 8
1.3.3. Contributions to the Field of Model-Based Diagnosis . . . 9
1.4. Thesis Outline . . . 10

2. Preliminaries 13

2.1. Recommender Systems . . . 14
2.1.1. Collaborative Recommender Systems . . . 15
2.1.2. Content-Based Recommender Systems . . . 18
2.1.3. Knowledge-Based Recommender Systems . . . 19
2.1.4. Hybrid Recommender Systems . . . 22
2.2. Configuration Systems . . . 24
2.3. Consistency Management . . . 26
2.4. Summary . . . 29

3. Consistency Management in Recommender Systems 33

3.1. Example: Recommending Digital Cameras . . . 34
3.1.1. Requirements of the Customer . . . 36
3.1.2. Intermediate Representation . . . 36
3.2. Algorithm: GraphXplain . . . 37
3.3. Algorithm: FastXplain . . . 41

3.4. Algorithm: Boosted FastXplain (BFX) 45
3.5. Algorithm: Personalized FastXplain (PFX) 49
3.6. Algorithm: PersRepair 52
3.7. Algorithm: ReAction 57
3.8. Evaluation 62
3.8.1. Performance Evaluation 64
3.8.2. Evaluation of the Acceptance Probability (Precision) 76
3.9. Related Work 79
3.10. Discussion 80

4. Consistency Management in Configuration Systems **83**
4.1. Example: Configurable Bike 84
4.2. Algorithm: FastDiag 87
4.3. Algorithm: FlexDiag 91
4.4. Algorithm: PersDiag 92
4.4.1. Personalization Strategies 94
4.4.2. Identifying Personalized Diagnoses 97
4.5. Evaluation 101
4.5.1. Performance Analysis 102
4.5.2. Performance Evaluation 102
4.5.3. Evaluation of the Acceptance Probability (Precision) 108
4.6. Related Work 113
4.7. Discussion 114

5. D-fame: The Diagnosis Framework **117**
5.1. System Architecture 117
5.1.1. Input-Component 117
5.1.2. Algorithms-Component 120
5.1.3. Datastructure-Component 121
5.1.4. Output-Component 122
5.2. Using the framework 122
5.3. Summary 124

6. Conclusion and Future Work **125**
6.1. Conclusion 125
6.2. Overview of the Algorithms 126
6.3. Future Work 128

Bibliography **131**

List of Figures

1.1. Hitting Set Directed Acyclic Graph for the Running Example 4
1.2. Focus of this PhD Thesis 6

2.1. Taxonomy of Recommender Systems 16
2.2. Graphical Relationship between User and Items 16
2.3. Critiquing Recommender System 20
2.4. Steps during the Interaction with a Knowledge-Based Recommender System 22
2.5. Hybrid Recommender System 23
2.6. Supporting Customers with Minimal Conflict Sets 30
2.7. Supporting Customers with Repair Actions 31

3.1. Two-mode Network of Requirements and Products 38
3.2. One-mode Network of the Requirements 39
3.3. *FastXplain:* Identification of Minimal Conflict Sets 42
3.4. *BFX* (Boosted FastXplain) Identification of Minimal Conflict Sets 47
3.5. *PFX* (Personalized FastXplain) Identification of Personalized Conflict Sets 51
3.6. *PersRepair:* Identification of Personalized Diagnoses 55
3.7. *ReDiagnosis:* Execution Tree 58
3.8. *ReDiagnosis:* Identification of Personalized Diagnoses 61
3.9. Item-based Evaluation to calculate Minimal Conflict Sets with a Satisfaction Rate of 35% . 67
3.10. Item-based Evaluation to calculate Minimal Conflict Sets with a Satisfaction Rate of 50% . 69
3.11. Item-based Evaluation to calculate Minimal Diagnoses with a Satisfaction Rate of 35% . . 71
3.12. Item-based Evaluation to calculate Minimal Diagnoses with a Satisfaction Rate of 50% . . 74
3.13. Requirement-based Evaluation to calculate Minimal Conflict Sets with a Satisfaction Rate of 35% 74
3.14. Requirement-based Evaluation to calculate Minimal Conflict Sets with a Satisfaction Rate of 50% 75
3.15. Requirement-based Evaluation to calculate Minimal Diagnoses with a Satisfaction Rate of 35% 76

3.16. Requirement-based Evaluation to calculate Minimal Diagnoses with a Satisfaction Rate of 50% . . . 77
3.17. Screenshots of the Application for the PC User Study . . . 78
3.18. Decision Tree for the Consistency Management in Recommender Systems . . . 81

4.1. *FD:* Execution Tree . . . 89
4.2. *FastDiag:* Identification of Personalized Diagnoses . . . 92
4.3. *HSDAG:* Identification of Minimal Diagnoses . . . 93
4.4. *PersDiag:* Identification of the Preferred Diagnosis (based on utility values) . . . 98
4.5. *PersDiag:* Identification of the Preferred Diagnosis (based on similarity values) . . . 99
4.6. *PersDiag:* Identification of the Preferred Diagnosis (based on probability values) . . . 100
4.7. Evaluation of the *CLib Benchmark FS* . . . 104
4.8. Evaluation of the *CLib Benchmark PC* . . . 105
4.9. Evaluation of the *CLib Benchmark Bike* . . . 106
4.10. Evaluation of the *CLib Benchmark Renault* . . . 107
4.11. Constraint-based Evaluation of the *Broadband* Model (from 2 to 40 constraints) . . . 108
4.12. Constraint-based Evaluation of the *Broadband* Model (from 2 to 20 constraints) . . . 109
4.13. Evaluation of the Relevance of Diagnosis Cluster . . . 112
4.14. Decision Tree for the Consistency Management in Configuration Systems . . . 114

5.1. Architecture of the Diagnosis Framework *D-fame* . . . 118
5.2. Overview of Different Weight Calculations . . . 120

6.1. Decision Tree for the Consistency Mangement in Constraint-based Systems . . . 127

List of Tables

1.1. Table-Based Product Data Representation . . . 3
1.2. Contributions . . . 8

2.1. Table of the Relationship between User and Items . . . 17
2.2. Recommendation of a Weighted Hybrid Recommender . . . 24

3.1. Product Assortment of the Running Example . . . 35
3.2. Intermediate Representation of the Running Example . . . 37
3.3. Intermediate Representation including Weights . . . 46
3.4. Utility Values specified by the Customer . . . 49
3.5. Utility Vlaues for each Product . . . 50
3.6. Similarity Value for each Product . . . 54
3.7. Overview of Performance Issues of the Introduced Algorithms . . . 65
3.8. Recommendation Task with one Minimal Conflict Set containing n Constraints . . . 65
3.9. Recommendation Task with one Minimal Conflict Set containing one Constraints . . . 66
3.10. Algorithm Overview to Calculate One Minimal Conflict Set . . . 68
3.11. Algorithm Overview to Calculate All Minimal Conflict Sets . . . 70
3.12. Algorithm Overview to Calculate One Diagnosis . . . 72
3.13. Algorithm Overview to Calculate All Diagnoses . . . 73
3.14. Precision Values for the *PC User Case Study* . . . 79

4.1. Log of the Past Configurations . . . 86
4.2. Utility Values specified by the Customer . . . 87
4.3. Similarity Values between the Requirements and the Log Entries . . . 96
4.4. List of Selected Diagnoses . . . 96
4.5. Algorithm Overview to Calculate All Diagnoses . . . 103
4.6. Overview of the Configuration Knowledge Bases . . . 103
4.7. Precision Values for the *PC User Case Study* . . . 110
4.8. Precision Values for the *Financial Service Case Study* . . . 111

7.1. Overview of the *WeCoTin* Configuration Knowledge Bases . . . 141

7.2. Evaluation of the *WeCoTin - Test Case 1* 142
7.3. Evaluation of the *WeCoTin - Test Case 2* 142
7.4. Evaluation of the *WeCoTin - Test Case 3* 143
7.5. Evaluation of the *WeCoTin - Test Case 4* 143
7.6. Evaluation of the *WeCoTin - Test Case 5* 144
7.7. Evaluation of the *WeCoTin - Test Case 6* 144
7.8. Evaluation of the *WeCoTin - Test Case 7* 145
7.9. Evaluation of the *WeCoTin - Test Case 8* 146
7.10. Evaluation of the *WeCoTin - Test Case 9* 146
7.11. Evaluation of the *WeCoTin - Test Case 10* 147

List of Algorithms

1. GraphXplain (M) 40
2. FastXplain(root, R, P) 44
3. BFX(root, R, P) 48
4. PFX(root, R, P, u) 52
5. PersRepair(R, P, n, c) 56
6. PersDiagnosis(R, NN, H, k) 57
7. ReAction(R, P, c) 59
8. ReDiagnosis(D, R, AR, P) 60

9. FastDiag(C $\subseteq$ AC, AC) 90
10. FD(D, C, AC) 90
11. PersDiag(C_R, C_{KB}, H, k) 100

Chapter 1

Introduction

Constraint-based systems are applied for solving problems that comprehend a set of conditions (constraints) which restrict the set of possible solutions. In order to solve such problems, constraint-based systems comprehend a wide range of techniques coming especially from Artificial Intelligence and Databases. Main application fields of constraint programming include scheduling and planning (Catillo, 2005) as well as recommendation (Jannach, 2008; Felfernig et al., 2009c) and configuration (Mittal and Frayman, 1989; Fleischanderl et al., 1998; Felfernig et al., 2004; Sinz and Haag, 2007). Most of these applications are interactive ones, meaning that a customer interacts with the system. Depending on the customers preferences, needs, and knowledge, the problem to be solved by the constraint-based system may become over-constrained (i.e. no available solution satisfies all customer requirements). In this situation intelligent techniques are needed to support the customer in finding a way to a consistent state (possible solution). For this reason, there is a need for a *consistency management*.

Consistency management can be applied in various types of constraint-based systems. For example, a *recommender system*, which is a system that guides a user in a personalized way to interesting or useful items, can be designed as a constraint-based system. *Configuration system* is another type of constraint-based systems. The aim of configuration systems is to aid customers in a special case of design activity to configure an item (product or service) which is assembled from instances of a fixed set of well-defined component types (Sabin and Weigel, 1998).

This chapter motivates the importance of consistency management in constraint-based systems. Furthermore, the research objectives and the contributions are pointed out in the field of recommender systems, configuration systems, as well as model-based diagnosis. An outline of the thesis closes this chapter.

1.1. Motivation

This thesis focuses on techniques and algorithms to aid customers in the conflict management process in constraint-based systems. While interacting with such systems, customers specify their preferences, requirements or needs in the form of constraints (called requirements in this work). In this context, situations

may occur where the specified requirements are too narrow, so that no solution can be found. Such a situation is also denoted as the *no solution could be found dilemma* (Pu and Chen, 2008).

In the following, a working example is introduced to show how a recommendation and a configuration task can be designed. A *recommendation* task is to find the best match between the customer requirements and the available items. Compared to this, a *configuration* task, is a design activity where the item that is configured, is created from a set of well-defined components (configuration knowledge base). The first step in such a configuration task is that the customer specifies the requirements. Afterwards these requirements are combined with the configuration knowledge base to retrieve at least one possible configuration.

For example, a customer wants to buy a bike and uses a constraint-based system to retrieve possible products. Let us assume, that the customer specifies the requirements a *blue*, *small*-sized bike with *18* gears. Furthermore, it is assumed, that only the colours *blue* and *green* are available for the bikes. Moreover, the bikes can be *small*, *medium* or *large*-sized. The gears shift is available with *3*, *10*, *12* and *18* gears. Additionally to these domain definitions, the following three constraints are introduced:

- c_1 : size $=$ *small* $\Rightarrow$ gear $<$ *12*,
- c_2 : colour $=$ *blue* $\Rightarrow$ size $=$ *medium*,
- c_3 : size $=$ *large* $\Rightarrow$ gear $\geq$ *12*,

Although this example is simple it characterizes a typical configuration task. This task consists of the constraints c_1, c_2 and c_3. As already mentioned, the requirements of the current customer are the following:

- r_1 : colour $=$ *blue*
- r_2 : size $=$ *small*
- r_3 : gear $=$ *18*

In comparison to a configuration task, which can be modelled as a constraint satisfaction problem (CSP) (Tsang, 1993), a recommendation task operates on a table-based product data representation. A configuration task can be transformed into a corresponding recommendation task. For the example described, a table may be set up which holds all possible configurations as product data (see Table 1.1). This table contains every possible instantiation of a bike which can be derived from the constraints (c_1, c_2 and c_3) and the domains (*colour:* blue, green; *size:* small, medium, large; *gears:* 3, 10, 12, 18). Each configuration in this table can be seen as a product with a corresponding identifier (see the column *id* in Table 1.1). Note that the number of products is also highly dependent on the domains. If, for example, one more colour is added, this results in 8 more products.

The aim of the customer, while interacting with a constraint-based system, is to identify products that satisfy all his requirements (r_1, r_2 and r_3). Considering the introduced example, it is not possible for the system to derive any bike that satisfies the requirements (r_1, r_2 and r_3). Taking a look at, for example, the requirements r_1 (colour = *blue*) and r_2 (size = *small*), it can be seen that they are conflicting with constraint c_2 (colour $=$ *blue* $\Rightarrow$ size $=$ *medium*). At the same time, it can be observed that there does not exist any product in Table 1.1 that satisfies the requirements r_1 and r_2.

Table 1.1.: Table-based product data representation of all possible bikes that can be derived from the constraints (c_1, c_2, c_3)

id	colour	size	gear
p_1	green	small	3
p_2	green	small	10
p_3	green	medium	3
p_4	green	medium	10
p_5	green	medium	12
p_6	green	medium	18
p_7	green	large	12
p_8	green	large	18
p_9	blue	medium	3
p_{10}	blue	medium	10
p_{11}	blue	medium	12
p_{12}	blue	medium	18

One approach to tackle this impasse, is to present an empty list of products to the customer. This is not convenient as the customer does not understand why no solution could have been found. Another approach is to inform the customer that there does not exist any product that satisfies their needs. Although this is slightly better than the first approach, it does not satisfy the customer. Therefore, applications are in need of techniques that support the identification of minimal sets of faulty requirements, which have to be deleted or adapted in order to restore consistency. This can be done by applying approaches and techniques from the field of model-based diagnoses (Reiter, 1987). An obvious approach is to focus on minimal cardinality diagnoses (Felfernig et al., 2004). A diagnosis can be defined as a set of requirements that need to be relaxed (or deleted) in order to retrieve at least one product (restoring consistency). The approach introduced by (Felfernig et al., 2004) focuses on minimal cardinality diagnoses, i.e. a set with a minimal number of requirement changes (relaxations) that can be used to restore the consistency.

Revisiting the working example, it has been already identified that the requirements r_1 and r_2 are conflicting with constraint c_2. This set of requirements $\{r_1, r_2\}$ is a minimal conflict set, because it is not possible to delete any element of it, in a way that a conflict is still induced. Another minimal conflict set of the example contains the requirements r_2 (size = *small*) and r_3 (gear = *18*). These requirements are conflicting with the constraint c_1 (size = *small* $\Rightarrow$ gear < *12*). (Reiter, 1987) introduced an approach to identify minimal diagnoses based on minimal conflict sets by building a Hitting Set Directed Acyclic Graph (HSDAG). This algorithm adds every minimal conflict set to the graph in an iterative way. The first minimal conflict set (r_1, r_2) is added to the root node (see Figure 1.1). The second minimal conflict set (r_2, r_3) is added to every leaf, if the path to this leaf does not contain any element of the minimal conflict set. In the working example, this holds for the left leaf (resulting from r_1), but it does not hold for the right leaf (resulting from r_2), because r_2 is also part of the second minimal conflict set. On the basis of this graph, all minimal diagnoses can be derived. For the working example the minimal diagnoses are $d_1 = \{r_2\}$ and $d_2 = \{r_1, r_3\}$. A minimal diagnoses is a relaxation of each minimal conflict set. Taking a look at the two conflict sets of the example, it can be seen that both sets include the requirement r_2. Thus, adapting the re-

quirement r_2 can resolve all minimal conflicts. Similar to this, if the requirements r_1 and r_3 are resolved, all minimal conflicts are resolved as well. Note that $\{r_1, r_2\}$ is also a diagnosis, nevertheless it is not minimal because $\{r_2\}$ is already a minimal diagnosis.

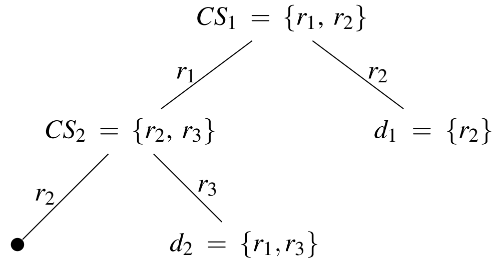


Figure 1.1.: Directed acyclic graph (as introduced by (Reiter, 1987) built of minimal conflict sets ($\{r_1, r_2\}$, $\{r_2, r_3\}$) to identify minimal diagnoses ($\{r_2\}, \{r_1, r_3\}$).

In the working example, the conflict sets have been identified without using a specific algorithm. Nevertheless, in a complex knowledge base it can be hard to identify minimal conflict sets. Therefore, Junker (Junker, 2004) introduced the algorithm *QuickXplain* to identify minimal preferred conflict sets. A *preferred* conflict set is more likely to be accepted by the user and thus, it is more valuable to the user compared to other conflict sets. The *QuickXplain* algorithm (Junker, 2004) uses a lexicographical ordering to define a preferred conflict set (also called explanation in (Junker, 2004)). Another approach (called *CorrectiveRelax*), that calculates *corrective* explanations was introduced by (O'Callaghan et al., 2005). Compared to (Junker, 2004) (where explanations are conflict sets), these corrective explanations introduced in (O'Callaghan et al., 2005) are similar to diagnoses. An example showing the mechanism of the *CorrectiveRelax* algorithm is given in Section 3.8.

It is already a valuable improvement to aid customers in the conflict management process using diagnoses. Nevertheless, this can still be improved by suggesting repair actions to the customers. A *repair action* is an action that can be performed by the customer in order to restore consistency. For the example introduced, the diagnoses $d_1 = \{r_2\}$ and $d_2 = \{r_1, r_3\}$ have been identified. Based on the diagnosis d_1, the system can suggest the customer to change the attribute *size* to *medium* in order to restore consistency. The suggested action (to change the attribute *size* to *medium*) is called repair action. A repair action for the diagnosis $d_2 = \{r_1, r_3\}$ is for example to change the *colour* to *green* and the *gear* to *10*. The number of alternatives could potentially become very large, which turns the identification of acceptable repair actions into a very frustrating task for the customer (Felfernig, 2007). One possibility to tackle this problem is to present only *representative* explanations to the customer. This representativeness ensures that the computed set of explanations is representative of all possible solutions (O'Sullivan et al., 2007). This can be achieved by reducing the number of explanations presented, and at the same time increase their diversity (see, for example, (O'Sullivan et al., 2007)). Another possibility is to decrease the number of explanations or repair actions by using personalization strategies. One possibility for a personalization strategy is to use similarities between different users or items. The similarity strategy tries to find the item that is most similar to the user requirements (Felfernig et al., 2009c). Another strategy is to introduce probabilities, and recommend the diagnosis or repair action with the highest probability of being accepted (Felfernig and Schubert, 2011a).

1.2. Research Objectives

Current research in constraint-based systems raises new challenges. The main challenges are performance, usability, personalization and conflict management. Among these issues, the improvement of run time performance, personalization and consistency management are especially relevant in the context of this work. In this context the following challenges have been tackled:

1. **Improving the run time performance for identifying minimal conflict sets by exploiting the structural properties of knowledge-based recommender systems:** As already described in Section 1.1 a recommender system operates on a table-based product assortment. The structural properties of such a table-based representation can be used to improve the run time performance of conflict detection algorithms.
   For interactive systems, it is important that the algorithms respond fast. According to (Miller, 1968; Card et al., 1991) the limits for a system to react should be: A system responding in less then **0.1 seconds** makes the user feel that the system reacts instantaneously. The upper limit to not interrupt the users flow of thought is **1 second**. Although the user will notice the delay of the system, there is no need for a special feedback if the response time of a system is between **0.1** and **1 second**. If the system takes longer then **1 second**, the user loses the feeling of operating on the data. Therefore, a fast response of interactive systems is crucial for user acceptance. For this reason, the approaches used for consistency management need to calculate the alternatives efficiently in order to not overcome the limit. Based on this time-bound and the possible high number of products, the following questions are addressed:
   (Q1) How can structural properties of knowledge-based recommender systems be used to improve the run time performance of conflict detection algorithms?
   (Q2) How can customers be supported in restoring consistency between the requirements and the corresponding product assortment?

2. **Improving the run time performance for calculating diagnoses** in configuration scenarios. For configuration systems with an interactive user interface, it is important that the algorithms respond fast (see Challenge 1).
   Compared to recommender systems, which use a table-based representation of the products or items, a configuration task can be modelled as a constraint satisfaction problem (CSP). As shown in Section 1.1, a configuration task can be transformed into a recommendation task. Nevertheless, this is in most cases not feasible, because the number of possible configurations is too large. For this reason, configuration systems are in the need of techniques and algorithms that aid customers in consistency management. Although there exists a couple of approaches to resolve conflicts that perform quite well (see for example (Junker, 2004; O'Callaghan et al., 2005)), there is still space for improvements. Therefore, this work focuses on the development of techniques and algorithms to calculate minimal diagnoses faster compared to existing approaches. This is the reason for raising the following question:
   (Q3) What are the possible run time improvements for diagnosis algorithms to support customers, who are interacting with a constraint-based system?

3. **Personalization strategies to improve the prediction accuracy:** A study performed by (Joachims et al., 2005) found out that 42% of the users clicked the top search hit, and 8% of the users clicked

the second hit. Similar results have been identified by other studies, but what is a valuable outcome of this study is that they performed a second test in which they secretly fed the search results through a script before displaying them to users. The script swapped the two top results so that what was originally the number two entry was displayed as the number one entry and vice versa. In this swapped display, 34% of the users still clicked on the top entry and 12% of the users clicked the second hit.

As the number of possible diagnoses resulting from an over-constrained situation gets potentially large, there is a need to intelligently rank the diagnoses. This leads to the following research question: (Q4) How can diagnoses be personalized in order to achieve a high prediction accuracy?

The research questions raised in this section indicate the basis for the objectives of this thesis. The following section provides an overview of the major contributions that have been developed within the scope of this thesis.

1.3. Contributions

The presented work focuses on consistency management principles in constraint-based systems, especially on knowledge-based recommender systems, as well as configuration systems. These systems are used to grasp the idea behind each algorithm. Nevertheless, the techniques and algorithms can also be applied to other over-constrained problems.

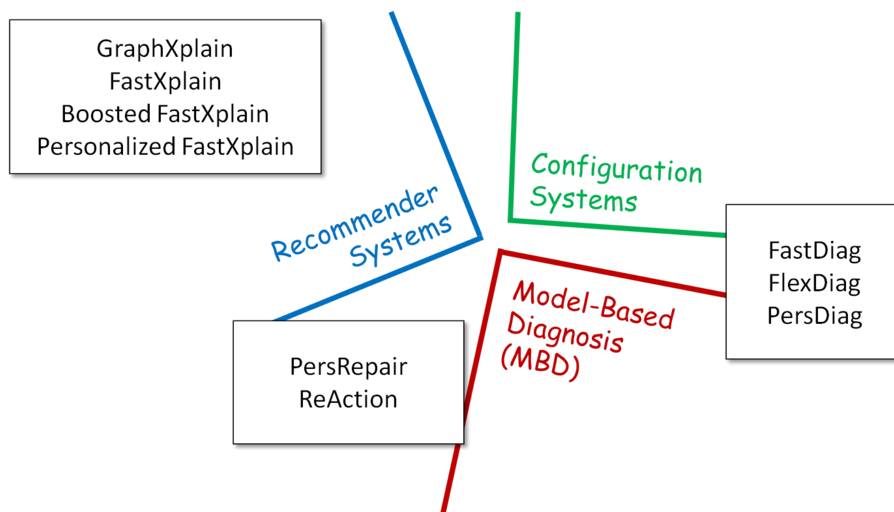


Figure 1.2.: This thesis focuses on consistency management techniques and algorithms for recommender and configuration systems, which also influence the research in the field of Model-Based Diagnosis (MBD)

This thesis covers algorithms and techniques from three fields: model based diagnosis (Reiter, 1987; de Kleer, 1990), recommender systems (Burke, 2000), and configuration (Soininen and Stumptner, 2003) (see Figure 1.2). Based on the needs of knowledge-based recommendation and constraint-based configuration systems, different algorithms have been developed to manage the consistency during the interaction with these systems. In the field of recommender systems we developed the algorithms *GraphXplain*,

FastXplain, *Boosted FastXplain* and *Personalized FastXplain*, which exploit the structural properties of predefined product catalogues. In addition, we developed the algorithms *PersRepair* and *ReAction* that strongly depend on concepts used in Model-Based Diagnosis (MBD). These two algorithms focus on different personalization strategies and are more independent from the underlying problem representation. In the field of constraint-based configuration systems we developed the algorithms *FastDiag*, *FlexDiag* and *PersDiag*. All three algorithms improve the consistency management and have an impact in the field of model-based diagnosis.

An overview of the achievements of this thesis, is given in Table 1.2. A more detailed discussion of the contributions obtained in the field of recommender systems are presented in Section 1.3.1. The achievements in the field of configuration systems are described in Section Section 1.3.2. The third field we contributed to is the field of model-based diagnosis. The influence of the presented work to the current research on model-based diagnosis is presented in Section 1.3.3.

1.3.1. Contributions to the Field of Recommender Systems

The contributions in the context of recommender systems focus on algorithms that exploit the structural properties of the table-based product data representation, as well as on algorithms that are more flexible regarding the underlying type of knowledge representation. The main contributions to the field of recommender systems are:

- **Application of concepts coming from network analysis to identify minimal conflict sets:** In (Schubert et al., 2009) the *GraphXplain* algorithm has been introduced to identify all minimal conflict sets based on common techniques used in network analysis (Wasserman and Faust, 1994). Starting from a table-based representation of the product assortment and the customer requirements a graph is generated. Afterwards, this graph structure is used to derive all minimal conflict sets.
- **Usage of heuristics to exploit the table-based product data representation:** In (Schubert et al., 2010) the *FastXplain* algorithm has been introduced, which is a heuristic to identify all minimal conflict sets from a given set of minimal diagnoses. A further improved algorithm is the *Boosted FastXplain (BFX)* which was introduced in (Schubert and Felfernig, 2011). This algorithm uses weights to optimize the selection of diagnoses used to calculate minimal conflict sets. Considering the fact that not only it is important to calculate minimal conflict sets efficiently, but also to personalize them. For this reason the *Personalized FastXplain (PFX)* is introduced. This algorithm uses utility values to identify relevant conflict sets for the user.
- **Improvement of the acceptance probability (precision):** Due to the primacy effect (Marshall and Werder, 1972; Glenberg et al., 1980) (the first few results are recalled more frequently) it is important to rank interesting diagnoses or repair actions at the top. Therefore, several algorithms have been developed within the scope of this research. First of all, the *Personalized FastXplain (PFX)*, which uses utility values to identify the most interesting results (conflict sets) for the customer. The *PersRepair* algorithm introduced in (Felfernig et al., 2009c) is an approach, that calculates personalized (plausible) repair actions for inconsistent requirements based on similarity measures. Moreover, the

Table 1.2.: Overview of the research questions and the corresponding contributions

Research Questions	Contributions
(Q1) How can structural properties of knowledge-based recommender systems be used to improve the run time performance of conflict detection algorithms?	The algorithms *GraphXplain* (Schubert et al., 2009), *FastXplain* (Schubert et al., 2010), *Boosted FastXplain (BFX)* (Schubert and Felfernig, 2011) and *Personalized FastXplain (PFX)* have been developed within this work. All algorithms use a table-based representation generated from the product assortment and the customer requirements.
(Q2) How can customers be supported in restoring consistency between the requirements and the corresponding product assortment?	Customers can be supported by an iterative presentation of minimal conflict sets (explicitly calculated by the algorithms *GraphXplain* (Schubert et al., 2009), *FastXplain* (Schubert et al., 2010), *BFX* (Schubert and Felfernig, 2011) and *PFX*), by the presentation of minimal diagnoses (explicitly calculated by the algorithm *PersDiag* (Felfernig and Schubert, 2011a)) or by the presentation of repair actions (explicitly calculated by the algorithms *PersRepair* (Felfernig et al., 2009c) and *ReAction* (Schubert et al., 2011)).
(Q3) What are the possible run time improvements for diagnosis algorithms to support customers, who are interacting with a constraint-based system?	There are several possibilities to improve state-of-the-art algorithms in terms of run time performance. The algorithms *ReAction* (Schubert et al., 2011) and *FastDiag* (Felfernig et al., 2011) are especially interesting, due to their general applicability. Another approach is the *FlexDiag* (Felfernig and Schubert, 2010b) which is a fast approach for calculating a small number of diagnoses. Nevertheless, it is not guaranteed that the algorithm finds minimal diagnoses.
(Q4) How can diagnoses be personalized in order to achieve a high prediction accuracy?	This work focuses on the application of different personalization strategies. The algorithm *PFX* uses utility values, whereas the algorithm *PersRepair* (Felfernig et al., 2009c) uses different similarity measures. Moreover, the algorithm *PersDiag* (Felfernig and Schubert, 2011a) is used to compare a utility, a similarity, a probability and a hybrid approach. A lexicographical ordering based on utility values is used by the algorithms *ReAction* (Schubert et al., 2011) and *FastDiag* (Felfernig et al., 2011).

ReAction algorithm (Schubert et al., 2011) was developed, which uses a lexicographical ordering based on utility values to identify the most suitable repair action for the customer.

1.3.2. Contributions to the Field of Configuration Systems

The contributions to the field of configuration systems address of the run time efficiency of algorithms to support users during the consistency management. On the other hand, the introduced concepts and

algorithms focus on personalization strategies to rank diagnoses. The main contributions to the field of configuration systems are:

- **Raising efficiency through personalization:** A customer selects higher ranked diagnoses or repair actions more often compared to others (primacy effect). If relevant diagnoses are ranked higher, then less evaluation effort is required from the customer. For this reason it is important to personalize the ranking of the presented diagnoses. In order to tackle this problem, the *PersDiag* algorithm (Felfernig and Schubert, 2010a, 2011a) has been developed to calculate personalized diagnoses. With this approach a comparison between different personalization strategies such as utility, similarity, probability and hybrids has been performed. Moreover, the *FastDiag* algorithm (Felfernig et al., 2010c, 2011) has been developed to identify personalized diagnoses using a lexicographical ordering.
- **Calculation of a limited set of minimal diagnoses:** Due to the primacy effect, there is no need to calculate all minimal diagnoses, just a few are enough. Therefore, the algorithm *FastDiag* (Felfernig et al., 2010c, 2011) is introduced, which allows an efficient calculation of one diagnosis at a time with logarithmic complexity in terms of the number of consistency checks.
- **Run time improvements:** For interactive systems, it is important that they can respond fast, because otherwise the customer loses the attention. For this reason, a detailed evaluation of the algorithms *FastDiag* (Felfernig et al., 2010c, 2011) and *PersRepair* (Felfernig and Schubert, 2010a, 2011a) has been performed and compared the results to state-of-the-art approaches such as *QuickXplain* (Junker, 2004) and *CorrectivRelax* (O'Callaghan et al., 2005).

1.3.3. Contributions to the Field of Model-Based Diagnosis

Model-Based Diagnosis (MBD) covers techniques and algorithms to explain faulty behaviour of a system. Based on the description of a system combined with an observation of the behaviour, the system can be identified to perform correctly (as it is meant to behave) or not (Reiter, 1987). If the description conflicts with the intended behaviour this results in a conflict situation. Then, the diagnosis task is to identify those components which explain the discrepancy between the observed and the intended system behaviour (Reiter, 1987). A *diagnosis* is a minimal set of faulty components whose adaptation will allow the identification of a recommendation or configuration.

Model-Based Diagnosis starts with a description of the system which is the predefined product assortment in recommender systems and the product knowledge base in configuration systems. The intended behaviour of the system is that it provides a recommendation or configuration for a given set of customer requirements. In cases where the system behaviour deviates (no solution can be found), one or more diagnoses can be calculated. The main contributions to the field of model-based diagnosis are the following:

- **Deriving minimal conflict sets based on minimal diagnoses:** Based on the table-based data structure which is used by recommender systems, diagnoses can be easily derived. These diagnoses are used by the algorithms *FastXplain* (Schubert et al., 2010), *Boosted FastXplain (BFX)* (Schubert and Felfernig, 2011) as well as *Personalized FastXplain (PFX)* to calculate minimal conflict sets. In order to derive these minimal conflict sets, an adaptation of the Hitting Set Directed Acyclic Graph (HS-DAG) (Reiter, 1987) has been performed. Originally minimal conflict sets are added to this graph

with the aim to identify minimal diagnoses. This approach has been adapted in a way that minimal diagnoses are added to the graph with the aim to derive minimal conflict sets.

- **Usage of the divide-and-conquer principle for calculating minimal diagnoses:** The divide-and-conquer principle has already been applied to identify minimal conflict sets (see *QuickXplain* (Junker, 2004)). Within this work two algorithms (*ReAction* (Schubert et al., 2011) and *FastDiag* (Felfernig et al., 2010c, 2011)) have been developed that use the divide-and-conquer principle for calculating minimal diagnoses. The focus on minimal diagnoses improves the run time behaviour considerably if only one or few diagnoses are needed.
- **Identification of diagnosis clusters:** A diagnosis cluster is a set of constraints of which at least one subset constitutes a minimal diagnosis. Nevertheless, the set (diagnosis cluster) itself does not need to be a minimal diagnosis itself. A diagnosis cluster allows us to quickly narrow the space where at least one minimal diagnosis is located without actually calculating the minimal diagnoses. In order to calculate such diagnosis clusters, the *FlexDiag* algorithm (Felfernig and Schubert, 2010b) has been developed. This algorithm takes one parameter which impacts the number of constraints that are part of the diagnosis cluster, but which are not part of the minimal diagnosis.
  The *FlexDiag* algorithm allows us to narrow the space where the minimal diagnosis is located without calculating the minimal diagnosis. For this reason, the algorithm can be applied in the model-based diagnosis community to identify rough clusters. This is especially useful in situations where no minimal diagnosis is needed or a further refinement is done by another approach.

1.4. Thesis Outline

This PhD thesis is partitioned into 6 chapters. First, an introduction to the field of research is given. Then the focus is put on consistency management techniques and algorithms, which can be applied in knowledge-based recommender and constraint-based configuration systems. Moreover, an overview of the *D-fame: Diagnosis FrAMEwork* is given that comprehends all techniques and approaches introduced. Finally, a conclusion and outlook on possible future work is provided. This thesis is organized as follows:

Chapter 1 outlines the motivation and research objectives for this work. It raises different research questions regarding consistency management in constraint-based systems such as recommender and configuration systems. An overview on the structure of this thesis completes this chapter.

Chapter 2 provides an overview on related work. Different constraint-based systems are introduced. This thesis concentrates on recommender systems as well as configuration systems. Section 2.1 gives an overview of different recommendation approaches such as collaborative filtering, content-based filtering and knowledge-based recommendation as well as hybrids thereof. Furthermore, the implementation of a recommendation system as a constraint-based system is sketched. Section 2.2 gives an overview of configuration systems. Finally, the concepts of inconsistency, consistency management and how to restore consistency are described in more detail (see Section 2.3). Additionally, basic definitions of terms such as *conflict* and *diagnosis* are introduced.

Chapter 3 presents the concepts of consistency management in the context of recommender systems. The discussions in Chapter 3 focus on specific algorithms to restore consistency in knowledge-based recommender systems. The discussed algorithms are *GraphXplain* in Section 3.2, *FastXplain* in Section 3.3,

BFX in Section 3.4, *PFX* in Section 3.5, *PersRepair* in Section 3.6 and *ReAction* in Section 3.7. All algorithms are evaluated against alternative state of the art approaches (Junker, 2004; O'Callaghan et al., 2005; Jannach, 2008) according to run time performance as well as prediction quality (acceptance probability). Section 3.9 further discusses related work from the field of consistency management in knowledge-based recommender systems. A short summary of the algorithms and a selection guide for certain situations concludes the chapter.

Chapter 4 presents the concepts of consistency management the context of configuration systems. The discussion focuses on algorithms to restore consistency in configuration systems. This chapter highlights three different algorithms, *FastDiag* in Section 4.2, *FlexDiag* in Section 4.3, and *PersDiag* in Section 4.4 to identify diagnoses or diagnosis clusters. All algorithms are evaluated against state-of-the-art approaches (Junker, 2004; O'Callaghan et al., 2005) according to run time performance as well as prediction quality (acceptance probability). An evaluation focusing on the relevance of the identified diagnosis clusters is included as well. A tentative summary of the algorithms completes the chapter. An evaluation focusing on the relevance of the identified diagnosis clusters is included as well. A tentative summary of the algorithms completes the chapter.

Chapter 5 provides an overview on the *D-fame: Diagnosis FrAMEwork*. This framework was designed to support the scientific community especially in the field of model-based diagnosis. Its aim is to provide a framework for evaluation, study and analysis of different algorithms for calculating minimal conflict sets and diagnoses. An overview on the architecture of the implementation is given. This chapter is concluded with a guideline on how to use the framework.

Chapter 6 concludes this thesis and reflects on the goals and contributions. In addition, an outlook on several possibilities for future research is given.

Chapter 2

Preliminaries

Today, people are constantly exposed to constraint-based systems in their everyday life. They appear in many different applications, spanning from time tabling, production scheduling (Catillo, 2005), and recommendation (Jannach, 2008; Felfernig et al., 2009c; Burke et al., 2011) to product and service configuration (Mittal and Frayman, 1989; Fleischanderl et al., 1998; Felfernig et al., 2004; Sinz and Haag, 2007). When interacting with constrainted-based systems, customers are asked to specify their requirements via constraints. These constraints can be combined with ones from other sources, for example, a knowledge base. All together, the constraints state the problem (a CSP, Constraint Satisfaction Problem) which can be solved by a constraint solver. Definition 1 defines a constraint satisfaction problem according to the work of (Tsang, 1993).

Definition 1 *A **Constraint Satisfaction Problem** (CSP) can be defined as a tuple $\langle V,D,C \rangle$, where **V** is a finite set of variables. Each of these variables is associated with a finite domain **D** and a set of constraints **C**. These constraints restrict the possible variable assignments.*

The task of a constraint solver is to assign a value to each variable while satisfying all constraints. A well known example of a constraint satisfaction problem is the *N-queens problem*. The task is to place N queens on a $N \times N$ chess board. The criteria is that any two queens are not allowed to threaten each other (i.e. both are on the same row, column, or diagonal line of squares). This example shows, how a constraint can be used to restrict the number of alternatives. Without the constraint, the queens could be placed anywhere on the board. If the constraint is added, there exists only a small number of placement alternatives, in which the N queens could be placed.

"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." (Freuder, 1997). This is the reason why constraint technologies are applied in many applications (Tsang, 1993), such as planning and scheduling (Catillo, 2005), recommendation (Felfernig et al., 2009c) and configuration (Mittal and Frayman, 1989; Fleischanderl et al., 1998; Sinz and Haag, 2007). The field of applications for planning and scheduling, for example, cover resource planning as well as production scheduling. Further example applications are software feature selection (Raatikainen et al., 2005), software release planning and

requirements engineering (Goknil et al., 2008; Felfernig et al., 2010d). The application domains for recommendation and configuration technologies are, for example, automotive industry, aviation and financial services. Constraint technologies can also be applied in groupware systems as shown in (Felfernig et al., 2010e).

The main advantage of constraint-based systems is their ability to reduce errors and lower response times. To illustrate the advantage, consider solving a Sudoku puzzle. Depending on the skills the person has, an error may be made, because it is hard to keep track of all restrictions to number placement. The constraint solver, on the other hand, keeps all constraints in the memory and reliably finds a correct solution. Additionally, if a Sudoku puzzle is solved on the computer, it can helpfully inform the user about erroneous placements before he realizes the conflict, avoiding (mentally) costly backtracking steps. In applications that use constraint technologies, many scenarios may occur where the constraint set can become inconsistent.

An inconsistent situation may emerge, when a customer interacts with a configurator application. While the customer refines her requirements (represented as constraints), they can become inconsistent with the underlying configuration knowledge base (O'Sullivan et al., 2007). Another application field of consistency management is the development and maintenance of knowledge bases. There are several possibilities that cause inconsistencies in knowledge bases. First, an inconsistency can be based on contradicting constraints. Another possibility is that the the knowledge base becomes inconsistent with respect to a set of test cases (Felfernig et al., 2004). Moreover, utility constraints (also called scoring rules) which determine the order in which configurations or products are presented, may get inconsistent with ranking examples coming from an expert (Felfernig et al., 2010a). These examples show, that there exists a need of intelligent assistance to actively support users and knowledge engineers.

This thesis is based on three pillars: model-based diagnosis (Reiter, 1987; de Kleer, 1990), recommender systems (Burke, 2000) and configuration systems (Soininen and Stumptner, 2003). The focus is to restore consistency by extending and adapting concepts of model-based diagnosis to calculate minimal conflict sets and preferred (plausible) minimal diagnoses (including repairs). The algorithms described can be used to improve the run time performance and prediction quality of constraint-based systems.

The remainder of this chapter is organized as follows. Section 2.1 gives an overview of different recommendation approaches. They are divided into three main categories which comprise collaborative recommender systems (Section 2.1.1), content-based recommender systems (Section 2.1.2) and knowledge-based recommender systems (Section 2.1.3). Section 2.2 introduces configuration systems and highlights their main challenges. The importance of consistency management is elaborated on in Section 2.3. Additionally, that section discusses different aspects of model-based diagnosis. A summary given in Section 2.4 closes this chapter.

2.1. Recommender Systems

Which book should I read? What mobile phone should I buy? What music should I listen to? In everyday life situations we trust recommendations from friends, random people and reviewers, but also from applications (for example recommendation web services). In the last years recommender systems have

experienced a significant upturn and can be found in many applications. The roots of recommender systems trace back to different fields comprehending cognitive science (Rich, 1979), information retrieval (Salton, 1988) as well as management science (Murthi and Sarkar, 2003). Especially the personalization issues were studied in these areas. In the mid-1990s recommender systems, an independent research area emerged. The break-through happened, when researchers started to focus on recommendation problems that explicitly rely on rating structures (Adomavicius and Tuzhilin, 2005).

The aim of recommender systems is to present new or desired items and information to the user. Presenting the right item at the right time is increasingly playing a key role in achieving a good usability. Different characteristics of products and services (for example, frequency of purchase, customizability, cost) are the reason why different types of recommender systems emerged. Nevertheless, all types of recommender systems try to find items that are of interest for the user. In order to achieve this goal, they apply different information filtering techniques. In order to grab the interest of the user, the recommender system has to build a user model using any kind of information it can retrieve, such as interactions of the user with the system (for example, ratings or purchases) or the similarities between users. For the recommendation the user profile is compared to some reference criteria. The possible criteria may be the description of an item (the content-based approach) or the user's social environment (the collaborative filtering approach) (Konstan et al., 1997).

In the context of this work a recommender system is defined as in (Felfernig and Burke, 2008):

Definition 2 *A* ***recommender system*** *is a system that guides the user in a personalized way to interesting or useful objects in a large space of possible options or that produces such objects as output.*

The main challenge of recommender systems is that, if there exists U users and I items, then the system has a state space of potential recommendations of $U \times I$. This state space is potentially too large for the user and therefore, we are in need of intelligent methods to rank and personalize the result. One approach for personalization is to identify user preferences. These preferences can be either retrieved *implicitly* by observing the users and their interactions with the system, or *explicitly* through a dialogue with the user. After having a model of the users preferences it can be used for different personalization strategies.

Recommender systems can be categorized into three main types (see also Figure 2.1): collaborative, content-based and knowledge-based recommender systems. Section 2.1.1 introduces the *collaborative filtering* approach which can be further divided into user-based and item-based ones. These subcategories express whether the focus is on the similarity between users or between items. In Section 2.1.2, an introduction to *content-based* recommender systems is given. These systems focus on the content of items that should be recommended. This content can be either knowledge about the items or contextual knowledge (context, for example, a web page in which the item occurs). Section 2.1.3 introduces *knowledge-based* recommenders which exploit deep knowledge about customer requirements and the underlying set of products. Knowledge-based recommenders can be further categorized into *constraint-based* and *case-based* (see Section 2.1.3 for a more detailed description).

2.1.1. Collaborative Recommender Systems

Collaborative recommender systems (also called collaborative filtering recommender systems) are based on the idea that a similar interest of users in the past is an indicator that these users will be interested in

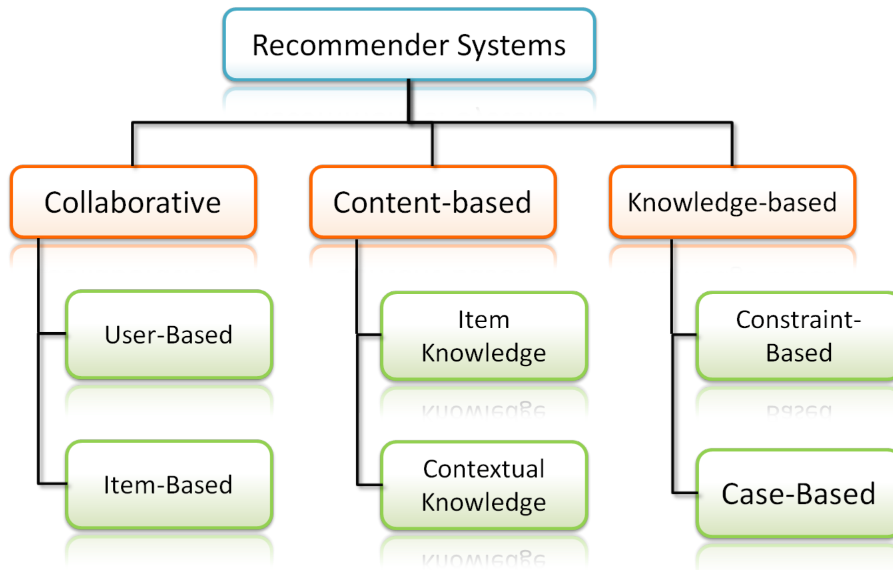


Figure 2.1.: Taxonomy of recommender systems based on the work of (Jannach et al., 2011) and (Ricci et al., 2011)

the same or similar items in the future. Collaborative filtering was the first type of recommender systems. The first papers describing methods and techniques of collaborative filtering emerged in the mid-1990s (Resnick et al., 1994; Hill et al., 1995; Shardanand and Maes, 1995). The first systems that emerged were *Tapestry*, a mail filtering system developed from Xerox Parc (Goldberg et al., 1992), and *GroupLens** a text filtering system (for example, for news articles) (Resnick et al., 1994). One of the first applied music recommenders using Pearson Correlation was presented by (Shardanand and Maes, 1995).

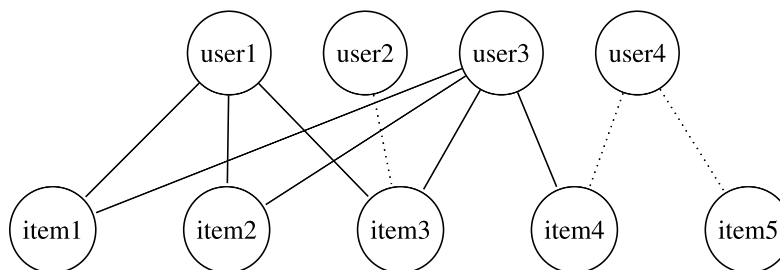


Figure 2.2.: Graphical relationship between users and items in a collaborative recommender system

The key idea of collaborative recommender systems is to use the social environment of users as a basis for recommendations. The focus is on a user model which is based on user interests. The user model can be generated by observing the user and accumulating the items they are buying or visiting frequently. On the other hand, interests can be explicitly stated by the user, for example, by providing a rating for each item. The interest vector from the user model can be used to relate items and users. A simple model that connects users and items is presented in Figure 2.2. This figure includes 4 users and 5 items. Note that this

*The homepage of the GroupLens research group is still active and can be found at `http:\www.grouplens.org`

figure does not include any information regarding a user's item preferences. For example *user1* rated the items *item1*, *item2* and *item3* and *user3* rated the items *item1*, *item2*, *item3* and *item4*. These two users can be considered to be similar to each other due to the fact that they rated similar item sets (3 out of 4 are the same). In this example, *item4* can be recommended to *user1* because the similar user (*user3*) has already shown interest in this item. Note that in this simple example, the actual rating value is neglected.

The relation between users and items can also be presented in a table (see, for example, Table 2.1). The values in this table indicate item ratings. For example, *user1* rated *item1* with a value of **3**, *item2* with a value of **4** and *item3* with a value of **3**. *User1* did not provide any rating for *item4* and *item5*. If a user did not provide any rating for an item this field is empty.

Table 2.1.: Table representation of the relationship between users and items in a collaborative recommender system. The values in this table are ratings from **5** (very good) to **1** (very bad). Based on these ratings, *user1* and *user3* are similar, as well as *item1* and *item3*.

	item1	*item2*	*item3*	*item4*	*item5*
user1	3	4	3		
user2			5		
user3	3	3	3	5	
user4				2	3

In collaborative recommender systems two techniques are used for recommendation (see also Figure 2.1). One way is to exploit the user similarity. This is called the *user-based* approach. In user-based collaborative filtering, the k-nearest neighbours of the active user are identified. The ratings of these users are exploited to predict the rating of the active user for a specific item. Let us consider *user1* (see Figure 2.2 and Table 2.1) as a current user. Considering the rating of this user (see Table 2.1), it can be observed that *user1* rated *item1* and *item3* with a value of **3** and *item2* with a value of **4**. Similar to this, *user3* rated *item1*, *item2* and *item3* with a value of **3** and *item4* with a value of **5**. Based on the ratings in Table 2.1, *user1* and *user3* can be considered as similar. As *user3* rated *item4* very well and the two users (*user1* and *user3*) are similar, *item4* may be recommended to the current user *user1*. Additionally to this user-based approach, a second collaborative filtering approach emerged, the *item-based* approach. This approach takes the similarity between items into account (Sarwar et al., 2001). If the items in Table 2.1 is compared with each other, it can be observed that *item1* got twice the rating **3** (from *user1* and *user3*). Moreover, *item3* got the same ratings from the same users and additionally the rating **5** from *user2*. Based on these ratings, *item1* and *item3* can be considered as similar. This similarity can now be used for the recommendation. Due to the knowledge that *user2* was interested in *item3*, and that *item1* and *item3* are similar, *item1* should be recommended to *user2*.

Additionally to the type of data used for recommendation, collaborative recommender systems can also be differentiated into memory based systems and model-based systems. Memory based systems process all user ratings online and incorporate them into the recommendation algorithm. Model-based systems use an additional abstraction mechanism, such as clustering or singular value decomposition (SVD) (Sarwar et al., 2002).

Collaborative recommender systems are very well suited for recommending items that are frequently used, such as music, movies[†], and books. The main advantage of this kind of recommender systems is, that there is no specific information about the items required, though the history of user interactions and user data (i.e. demographic data), has to be stored. Additionally, collaborative recommender systems require user ratings, which indicate the users interest.

Summarizing, collaborative recommender systems are the most wide spread type of recommender systems. Their advantages and limitations are well understood. Excellent overview papers on collaborative filtering are (Herlocker et al., 2004; Adomavicius and Tuzhilin, 2005; Anand and Mobasher, 2005; Smyth et al., 2005) and (Schafer et al., 2007).

2.1.2. Content-Based Recommender Systems

Content-based recommender systems aim to help users to deal with information overload. Therefore, content-based systems focus on the characteristics of the items. Compared to this, collaborative filtering recommender systems rely on user ratings. One advantage of collaborative filtering is that it does not need to provide up-to-date information about the items, which is costly. Nevertheless, collaborative filtering lacks an intuitive way of recommending items. For example, it would be obvious to recommend the new *Jamie Oliver* book to the user U_1, if the system knows that (1) the book is a cooking book and that (2) the user U_1 likes cooking and is interested in new books within this genre. Even though such a recommendation is straight forward, it is not supported by collaborative filtering techniques. For such a recommendation two types of knowledge are necessary: first, a description of the item and second, a user profile. If this knowledge about the item and the user is available, the recommendation task is to find the best match between items and a user profile (Pazzani, 1999). This approach is called content-based recommendation. The main advantage of content-based recommendation is that it does not require large user communities. Nevertheless, it needs information about the items and their characteristics. This knowledge about the item is often seen as *content*. For this reason this type of recommender systems is called *content-based* recommender systems.

Content-based recommender systems emerged from information retrieval. The recommendation methods and techniques are based on the ones that are used in the field of information retrieval and information filtering. The best known concepts and techniques are: TF-IDF (term frequency and inverse document frequency), vector space document model, feature selection, neural networks, k-nearest neighbour, naive Bayes text classification, as well as other classification techniques (for further descriptions see (Salton and Buckley, 1988; Baeza-Yates and Ribeiro-Neto, 1999; Adomavicius and Tuzhilin, 2005; Pazzani and Billsus, 2007)). Feature selection is another method that is used in content-based recommender systems. This feature selection is also known as variable selection or attribute selection and can be applied to select a subset of relevant features for building a robust learning model. Most methods using feature selection emerged from the system *WordNet*[‡], see especially (Pazzani and Billsus, 1997). (Manning et al., 2008) and (Chakrabarti, 2002) give an overview of different techniques. A comparison of which learning-based tech-

[†] The Netflix Price is the best known challenge that aims at the improvement of algorithms for recommending movies. See also `http://www.netflixprize.com`

[‡] WordNet is a research system developed at the Princeton University. It provides a semantic taxonomy for English vocabulary. More information can be found at `http://wordnet.princeton.edu/`

niques are suited especially for content-based recommender systems is presented by (Pazzani and Billsus, 1997).

The advantage of content-based recommender systems is that they can deal with a large amount of text-based information. For this reason, they are well suited for sites with a large amount of texts (for example, portals, emails) as well as domains with items which are content-rich (for example, news or books). The main drawback of content-based recommendation is that it depends on the underlying text quality. Similar texts written in the same language using the same keywords can be of good but also of bad quality. In addition, content-based approaches are not effective in small specific sites incorporating only small amounts of text. In contrast to collaborative filtering, content-based approaches cannot establish novel connections. It may happen, that in certain problem domains, too similar items are presented which are not of interest to the customer. For example, the aggregations of news articles, because the same story is published in several articles.

In the literature there is often no exact distinction between *content-based* and *knowledge-based* recommender systems. However, this work follows the classification where content-based recommender systems focus on the description of items. Compared to this knowledge-based recommender systems exploit some additional information, such as means-end knowledge (Felfernig and Burke, 2008; Ricci et al., 2011; Jannach et al., 2011).

2.1.3. Knowledge-Based Recommender Systems

Collaborative (see Section 2.1.1) and content-based (see Section 2.1.2) recommender systems have their advantages and strengths. Nevertheless, there are situations where these technologies are not applicable. Typically, customers do not buy a car, a house or a washing machine very frequently. Especially for these high-involvement products (for example, car or house) collaborative and content-based recommender approaches are less applicable. Due to the low number of available ratings a pure collaborative approach would fail at successfully recommending one of the items (Burke, 2000). This is based on the fact that, the sparse number of co-occurring ratings does not allow for a statistically sound calculation of neighbourhoods. (Bell and Koren, 2007) propose to use at least 100-nearest neighbours in their collaborative approach. It would also take a rather long time to gather the ratings, which is also a drawback of these approaches. Additionally, the time-span between two sessions of the user is important. A rating that is 3 or 5 years old is not valuable because user preferences evolve over time. Another point that cannot be handled by collaborative and content-based recommender systems, is that for more complex (and more expensive) products, customers often want to specify their preferences explicitly. A typical customer requirement from the domain of cars would be for example: *"the car should be a combi"* or *"the colour of the car should be blue"*. An explicit formulation of such preferences is not possible in purely collaborative and content-based recommender systems. Especially in these domains where traditional recommenders are not really suitable, knowledge-based recommenders emerged (Burke, 2000; Ricci et al., 2003; Felfernig and Burke, 2008). Knowledge-based recommenders allow users to explicitly specify their requirements. They also incorporate a deep knowledge about the underlying product assortment.

Knowledge-based recommender systems can be categorised into the following two types (Jannach et al., 2011): *constraint-based* (Thompson et al., 2004; Felfernig and Burke, 2008; Zanker et al., 2010) and *case-based* (Burke, 2000; Bridge et al., 2005). The interactions of both types are similar. The customers are

specifying their requirements and the system tries to find a recommendation for the given preferences. The system may provide explanations for the recommended items. The main difference between the two types of knowledge-based recommender systems is the way they use the knowledge about the product items. *Constraint-based* recommender systems rely on explicitly defined recommendation rules (constraints) to identify a recommendation. In comparison, *case-based* recommender systems recommend items that are similar. Various similarity measures (see, for example (Konstan et al., 1997; Wilson and Martinez, 1997; McSherry, 2004)) are used for the recommendation. Another characteristic of traditional case-based recommender systems is that users (re-)specify their requirements in a query-based form until the target item, which satisfies the customer is found. If the customer is not an expert of the domain, this can lead to numerous interactions, until he finally finds a feasible item (Burke, 2002b). This drawback was the reason for the development of *critiquing* systems (Hammond et al., 1995). These critiquing systems are also case-based recommender systems, and what makes them special is that the navigation is supported by *critiques*. An example system is sketched in Figure 2.3. This system aids customers in find a digital camera. In each step one camera is presented to the customer. This camera can be critiqued using one of the attributes (price, mega pixel, display size, weight or optical zoom). In this critique the customer can either choose more of the attribute (for example, a larger display) or less of the attribute (for example, a lower price). In this way customers specify their requests in form of goals that are not satisfied yet (Burke et al., 1997).

Figure 2.3.: Critiquing recommender systems. For finding a suitable camera the customer can *critique* the current product. In the system displayed, the following attributes can be critiqued: *price*, *mpix*, *display*, *weight* and *optical zoom*

One main advantage of knowledge-based recommender systems is that there does not exist a *cold start problem* (also called ramp-up problem) (Burke, 2000; Felfernig and Burke, 2008) as this type of recommender systems use deep knowledge about the product assortment and the customer requirements. Knowledge-based recommender systems may achieve a great precision and good transparency; for example, by explaining the recommendation to the customer. Especially the transparency improves the trust of

the customer. On the other hand, for such systems it is required to create and maintain the knowledge. To deal with this challenge, domain experts and knowledge engineers have to invest considerable time efforts in order to develop and maintain the knowledge bases and keep them up-to-date. This is called the *knowledge acquisition bottleneck*. Additionally to these technical challenges, it is important to consider consumer decision making strategies in the design to improve the quality of the recommendation process and to increase customer satisfaction (Mandl et al., 2011). Research in this field has shown that the format of the information presented influences the behaviour of the customers. The way of presenting the recommendation highly influences the decision making strategy of the customers (see, for example, (Asch, 1949; Payne, 1976; Bettman and Kakkar, 1977; Lussier and Olshavsky, 1979; Tversky and Kahneman, 1981)). Thus it is important to focus not only on the algorithms, but additionally incorporate a suitable user interface and information presentation.

In order to retrieve a recommendation from a knowledge-based recommender system the customers have to specify their requirements. In a typical interaction, the recommender guides the customer (repeatedly) through the following phases (see also Figure 2.4):

- **(Phase 1)** *Requirement elicitation phase*: In the first phase of the interaction the customers identify and specify their preferences and requirements. Examples for requirements from the financial service domain are: *"I want to invest my money for 3 years"* or *"I want to invest my money for my children"*. The requirement elicitation phase is often realized through a set of predefined questions that the customers are supposed to answer. In this phase the system can include different methods to support or influence the customer by using for example default values (Huffman and Kahn, 1998).
- **(Phase 2)** *Suitability check*: In this phase the recommender application checks whether there exists a product that satisfies all requirements of the customer. If the application is not able to find a solution, there are different possibilities to fail gracefully. For aiding the customer in dealing with this inconsistency (no solution could be found), different techniques have been developed. One opportunity is to present a set of changes to the customer and if these changes are accepted a recommendation can be guaranteed. This changes can be, for example, *"Change the investment period to 5 years"* or *"Lower the interest rate to 3 %"*.
- **(Phase 3)** *Result presentation*: In the third phase the application presents the recommendation of product(s) if all requirements of the customer can be fulfilled. The alternative products are typically ranked using, for example, some kind of utility or similarity measures.
- **(Phase 4)** *Explanations*: For each recommended product the customer can ask the system to provide an explanation why this product was recommended. An explanation typically consists of an argumentation that relates the properties of the product to the specified requirements of the customer.

When customers specify their requirements, situations can occur where no product of the assortment fulfils all of these requirements. This situation is called the *no solution could be found dilemma* (Pu and Chen, 2008). It is rather disappointing for the customer if no adequate support is available to get out of this situation. In order to aid customers with restoring the consistency, existing approaches focus on low-cardinality diagnoses (Junker, 2004; Felfernig et al., 2004; Jannach and Liegl, 2006). These approaches identify sets with a low number of requirement changes (called diagnoses) that can be used to restore consistency. Similar to this, (O'Sullivan et al., 2007) introduced an approach to identify representative explanations to aid customers. The representativeness introduced in (O'Sullivan et al., 2007) ensures that

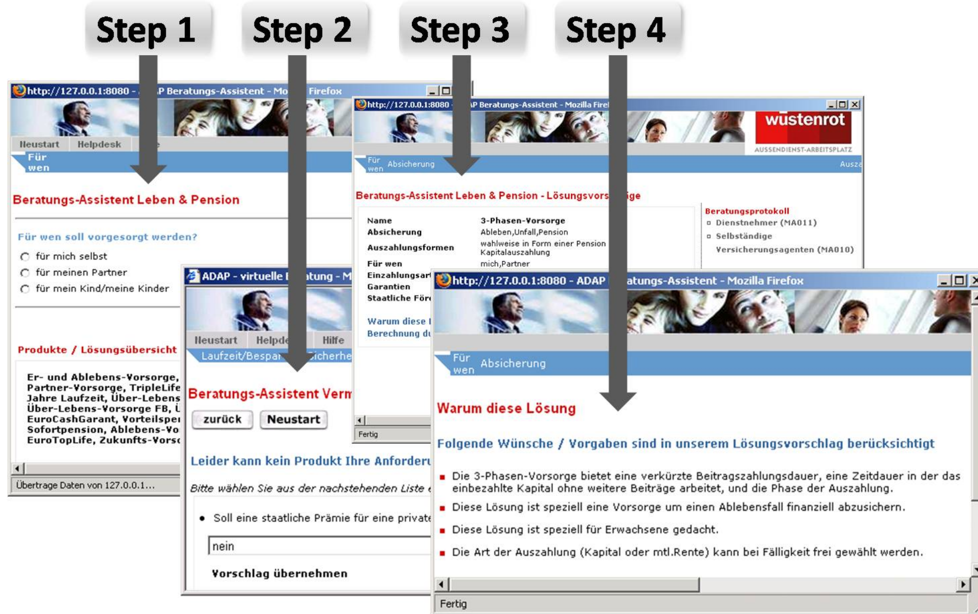


Figure 2.4.: Four step process in a knowledge based recommender system: *Phase 1*: requirement elicitation, *Phase 2*: suitability check, *Phase 3*: result presentation and *Phase 4*: explanations

the computed set of explanations is representative of all possible maximal relaxations and exclusion sets. In other words, a set of explanations is representative if it contains a relaxation (an exclusion set) containing each constraint that appears in a relaxation. The explanations contain a set of constraints that can be satisfied (relaxation) and a set of constraints that must be excluded (exclusion set) according to (O'Sullivan et al., 2007). Other approaches focus on the identification of maximally successfully sub-queries (Godfrey, 1997; McSherry, 2004; Jannach, 2008). A maximally successfully sub-query is based on the original query and keeps as many requirements as possible in order to still find a feasible item. In Chapter 3 different consistency management principles in recommender systems are introduced and discussed in detail.

Summarizing, knowledge-based recommender systems are very well suited for one time buyers. Most people buy products such as dish washers, washing machines, digital cameras, computers, cars, houses, ... only once in years and thus do not have a deep knowledge about such items. In addition, there is no possibility to construct user profiles for each user due to the few interactions over time. Nevertheless, user profiles should be established in order to enable a personalization of the recommendation. Constraint based recommender systems become important when there are specific requirements of the user that the solution has to meet (Felfernig and Burke, 2008). This thesis focuses on inconsistent requirements and discusses ways about how consistency between customer requirements and the product assortment can be attained.

2.1.4. Hybrid Recommender Systems

The most prominent types of recommender systems are: collaborative (see Section 2.1.1), content-based (see Section 2.1.2) and knowledge-based (see Section 2.1.3) ones. All approaches have advantages and disadvantages. The aim of hybrid recommender systems, is to combine different approaches in order

to overcome some of the drawbacks. The Netflix Price Competition[§] encouraged many researchers and students to combine different collaborative filtering techniques in order to improve accuracy. For example, the winner team of this competition used a weighted hybridization strategy.

A hybrid approach combines different base approaches for identifying a recommendation (see Figure 2.5). The input of a hybrid approach are various recommendation methods that focus on different knowledge sources. The hybrid approach combines the strengths of the base methods in order to overcome their shortcomings and problems. Similar to the base methods, the output of a hybrid approach is a ranked list of items - the recommendation list.

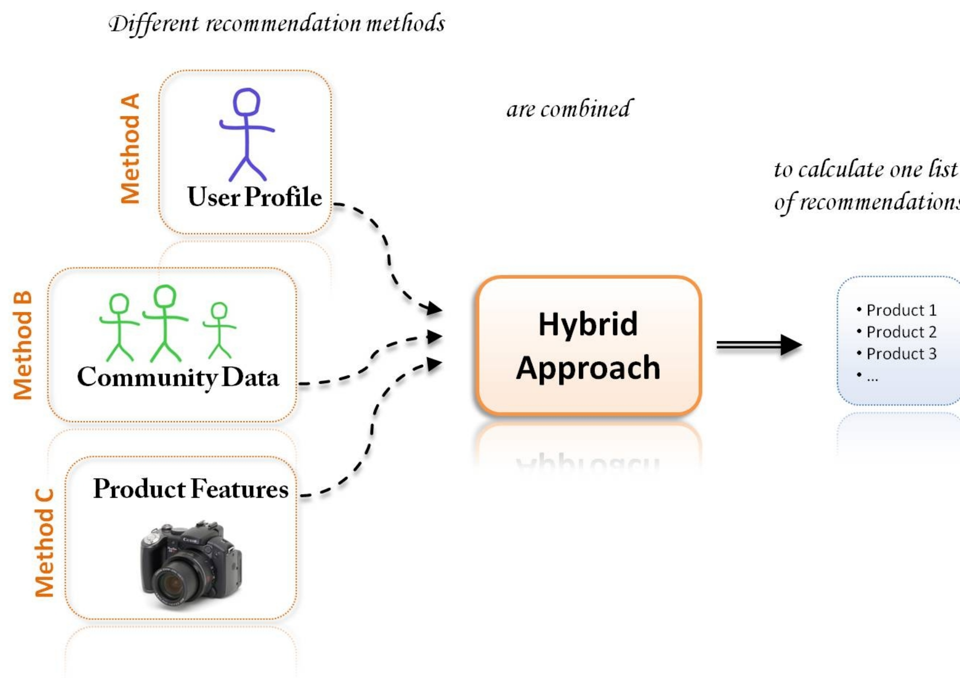


Figure 2.5.: A hybrid recommender system takes different recommendation methods (i.e. recommendations based on user profile, community data and product features) and combines them for calculating one list of recommendations

(Burke, 2002a) classified hybrids into seven different categories, to be specific: weighted, switching, mixed, feature combination, feature augmentation, cascade, and meta-level. The weighted approach, for example, combines scores from each individual recommendation algorithm using a linear formula. Let us assume, that two different recommendation techniques (**A** and **B**) should be combined. Due to the expert knowledge, it is known that the technique **B** normally performs better compared to **A**. For this reason, score weights are introduced. The scores for technique **A** are *0.4* for the top ranked item, *0.25* for the second ranked item and *0.1* for the third ranked item. Similar to this, the scores for technique **B** are *0.8*, *0.5* and *0.2* (see also Table 2.2). Note that there are several possibilities to estimate these score weights. (Zanker and Jessenitschnig, 2009), for example, conducted a sensitivity analysis to identify the optimal weighting schema. Another approach was introduced by (Claypool et al., 1999) which uses a dynamic weighting schema, based on the mean absolute error.

[§] http://www.netflixprize.com

For the recommendation of an item, both techniques (**A** and **B**) calculate their results separately. The result for technique **A** is *item1* (first rank), *item4* (second) and *item3* (third). And the one for technique **B** is *item2* (first rank), *item1* (second) and *item3* (third). These ranks are also shown in Table 2.2. Based on the calculated ranks, the scores which have been specified before, are arranged. Now the final score of each item can be calculated. This is done by summing up all scores for this item and dividing the result by the number of techniques which have been used. For example, the score of *item1* is *0.4* and *0.5*. The sum of these two values is *0.9*. This value is divided by *2* (as two techniques were used) which results in the final score of *0.45*. After determining the final score for each item, the weighted hybrid approach recommends the items in the following order: *item1*, *item2*, and *item3*.

Although there exist several hybridization strategies, none of them is applicable in all circumstances. Nevertheless, most of the base algorithms can be improved by using a hybrid approach. Several algorithm variants are presented in (Zanker et al., 2007). Many studies on hybrids have been performed (see for example (Balabanović and Shoham, 1997; Pazzani, 1999; Sarwar et al., 2000)).

Table 2.2.: Calculation of a weighted hybrid recommendation based the on ranks of the techniques **A** and **B**. According to the ranks, the scores are arranged. Finally the scores are combined by summarizing the scores of each item and dividing them by the number of techniques (2). The final score (Score **Hybrid**) leads to the final ranks (Rank **Hybrid**).

	Rank A	Score A	Rank B	Score B	Score Hybrid	Rank Hybrid
item1	1	0.4	2	0.5	0.45	1
item2		0	1	0.8	0.4	2
item3	3	0.1	3	0.2	0.15	3
item4	2	0.25		0	0.125	

2.2. Configuration Systems

Configuration is a sub-field of Artificial Intelligence with a lot of successful applications. One of the first configuration systems was the *R1/XCON* in the early 1980s (McDermott, 1982). After the introduction of this system the experiences and benefits have been widely studied (Barker et al., 1989; McDermott, 1994). Other successful applications of configuration systems that emerged in industry cover challenging products, such as telecommunication switches at Siemens (Fleischanderl et al., 1998) and other kinds of telecommunications products at AT&T (Wright et al., 1993; McGuinness and Wright, 1998). Moreover, configuration systems also became part of Enterprise Resource Planning systems, such as *Baan* (Yu and Skovgaard, 1998) and *SAP* (Haag, 1998, 2005). The description of configurable products (Soininen et al., 1998) have been studied and deployed in companies (Fleischanderl et al., 1998). Furthermore, a lot of other systems have emerged from research groups (for example *COSSACK* (Frayman and Mittal, 1987), *PLAKON* (Cunis et al., 1989, 1991), *COCOS* (Stumpner et al., 1994) and *WECOTIN* (Tiihonen et al., 2003)). In addition, a number of commercial configuration systems have been developed, for example

Trilogy SalesBUILDER (Hales, 1992) and *ILOG Configurator* (Mailharro, 1998; Junker and Mailharro, 2003).

This thesis also focuses on knowledge-based configuration systems. These knowledge-based configuration systems aim to build a complex system from simpler components. This system can be a complex product or a service (for example, telecommunication systems, computers, railways or financial services). Informally, *configuration* can be seen as a *special case of design activity, where the artefact being configured is assembled from instances of a fixed set of well-defined component types which can be composed conforming to a set of constraints* (Sabin and Weigel, 1998). The main characteristic of knowledge-based configuration systems is that they incorporate a deep knowledge of the products or services. These products or services can be configured using this knowledge. The knowledge can be formulated using constraints.

Configuration systems aid customers in configuring a product or service in a reasonable time which satisfies the specific needs of an individual customer. In order to combine an efficient production with satisfying individual needs, configurable products emerged (Sabin and Weigel, 1998). Product and service configuration systems are applied in mass customization (Davis, 1989; Pine, 1999; Tseng and Jiao, 2001) for addressing this challenge. Configuration systems are not limited to products such as cars, computers or washing machines, but also help with, for example, the configuration of feature models in software systems (Beuche et al., 2004; von der Maen and Lichter, 2004). Additionally to the correct behaviour and the hard real-time constraint (see Section 1.2), a well designed user interface helps the customers to deal with the complexity of alternatives (Felfernig et al., 2010b).

A configuration system typically has a configuration model. This model (also called configuration knowledge base) may contain rules about technical restrictions, the production processes and economic factors. Based on this model customers may configure products or services during an interactive session. A typical interactive configuration session consists of 3 main steps:

- (1) The **requirement specification** step: In this phase customers specify their requirements that should be satisfied by the configuration.
- (2) The **requirement adaptation** step: If there is no configuration possible that satisfies all specified requirements, then the customer has to adapt these requirements. The configuration system may aid the customer in this process using a consistency management technique.
- (3) The **result presentation** step: In this step all possible configurations that can be determined from the configuration knowledge base and the customer requirements are presented.

After the customers have specified their requirements (step 1), it may happen that no valid configuration can be found that satisfies all specified requirements. In such an over-constrained situation (step 2) it is difficult for the customer to choose successful modifications for the requirements (Felfernig et al., 2004). Due to the complexity of the configuration task, it is often not obvious which requirements are conflicting with the configuration model.

If an inconsistent situation occurs, the configuration system has to notify the customer that no solution exists. Such a situation is rather disappointing for the customer, especially if no further support is available to get out of this situation. In order to aid customers with restoring the consistency, existing approaches focus on low-cardinality diagnoses (Junker, 2004; Felfernig et al., 2004; Jannach and Liegl, 2006). These approaches identify sets with a low number of requirement changes (called diagnoses) that can be used to

restore consistency. Other approaches focus on the identification of maximally successfully sub-queries (Godfrey, 1997; McSherry, 2004). A maximally successfully sub-query keeps as many original customer requirements as possible in order to still find a feasible configuration (see Section 2.3 for a more detailed discussion). Chapter 4 introduces additional approaches.

2.3. Consistency Management

Constraint-based technologies (Tsang, 1993) are applied in different areas, such as recommendation (Felfernig et al., 2009c), configuration (Mittal and Frayman, 1989; Fleischanderl et al., 1998; Sinz and Haag, 2007) and scheduling (Catillo, 2005). There are many scenarios in which a constraint set may turn inconsistent. For example, when multiple actors with different opinions, views and interpretations try to find one common result. In this scenario, an inconsistent situation emerges, if the opinions or views are contradicting. Furthermore, an inconsistent situation may occur, if customers specify their requirements in a way that they are over-constrained (O'Sullivan et al., 2007). The system that tries to find a feasible product is not able to fulfil the task due to the inconsistency between the customer requirements and the product assortment. Moreover, an inconsistent situation may occur while implementing or maintaining a configuration knowledge base (Nguyen et al., 1987). In this scenario the configuration knowledge base may become inconsistent with a set of test cases (Felfernig et al., 2004). These examples demonstrate, that there exist several possibilities where an inconsistency can emerge. In order to actively support the customer in consistency management, intelligent techniques are needed.

An ideal tool for helping customers with consistency management, should help to establish, express and reason about the relationships between constraints (Finkelstein, 2000). Moreover, an ideal tool should check the consistency between these relations and detect the inconsistencies immediately. Additional information to help users to deal with an inconsistency as well as visualizations, can support consistency management. Beyond that, a functionality to track emerging inconsistencies is desirable.

Model-Based Diagnosis

Model-based diagnosis (MBD) (Reiter, 1987; de Kleer and Williams, 1987; Hamscher et al., 1992) aims to determine whether a system behaves correctly according to its predefined model. The field of model-based diagnosis emerged in the mid-1980s. The applications are diverse (see for example (Struss, 2002; Yongli et al., 2006)). This thesis applies and develops techniques for diagnosing faulty customer requirements in interactive constraint-based systems.

The aim of model-based diagnosis is to figure out, if a system behaves correctly according to its model. This model is a description of the system. Let us go back to the example introduced in Section 1.1. In this example, the *configuration model* consists of the product constraints (c_1, c_2, c_3), the domain of the variables, and the customer requirements (r_1, r_2, r_3). The *recommendation model* consists of the product assortment (see Table 1.1), and the customer requirements (r_1, r_2, r_3). Additionally to these models, a description of the correct behaviour is needed. Therefore, it is defined, that the behaviour is correct, if the system can find at least one configuration or recommendation that satisfies all customer requirements. In the case of the system behaving abnormally (no solution can be found), one or more diagnoses can be

calculated. A diagnosis consists of a minimal set of faulty components whose adaptation will allow the identification of a recommendation or configuration.

Several approaches have already been developed that aid customers in an inconsistent situation. State-of-the-art approaches calculate minimal diagnoses (McSherry, 2004; Felfernig et al., 2009c, 2011), maximally successful sub queries (Godfrey, 1997; McSherry, 2005), representative or corrective explanations (O'Sullivan et al., 2007; O'Callaghan et al., 2005), minimal conflict sets (Junker, 2004) or minimal unsatisfyable subsets of constraints (Liffiton and Sakallah, 2008). The different approaches are described in the following.

Customers can be aided with a minimal set of requirements (diagnosis) that need to be adapted or deleted in order to restore the consistency. In Section 1.1 an example has been introduced in which the minimal diagnoses have been derived from minimal conflict sets (using a HSDAG, Hitting Set Direct Acyclic Graph (Reiter, 1987)). The performance of this HSDAG technique heavily depends on the identification of minimal conflict sets. (Junker, 2004), for example, introduced an approach (*QuickXplain*) to identify such minimal conflict sets efficiently. The *QuickXplain* approach uses a divide-and-conquer strategy in order to map a given problem to a simpler ones (dividing principle). These simpler problems are then evaluated until a conflict has been identified (conquer-principle).

Another possibility to aid customers in an inconsistent situation is to present them product items or configurations that fulfil as many requirements (constraints) as possible. This can be achieved by identifying *maximal successful sub-query* (XSS). A possible maximal successful sub query for the example given in Section 1.1 is XSS_1: *colour = blue, gear = 18* (r_1, r_3). This is a maximal successful sub query, since it leads to at least one bike. Additionally, no requirement can be added to the sub query in a way, that the sub query still leads to a product (r_1, r_2, r_3 does not lead to any bike). Note that maximal successful sub queries are the complement of diagnoses. For example, XSS_1 (r_1, r_3) is the complement of diagnosis $d_1 = \{r_2\}$. (Jannach, 2008) presented an approach (*MinRelax*) to identify maximal successful sub queries by incrementally eliminating one or more constraints from the query of the customer (*query relaxation*). For the example, the query including all requirements does not lead to a solution. Thus the *MinRelax* algorithm eliminates the first requirement (r_1) and checks if the sub query including r_2 and r_3 leads to a bike. Due to the fact, that it is not possible to have a small-sized (r_2) bike with 18 gear (r_3), the second requirement (r_2) is eliminated from the original set of requirements. For this reason, *MinRelax* checks if it is possible to retrieve a blue (r_1) bike with 18 gears (r_3). This is possible and therefore, a maximal successful sub query has been found ($XSS_1 = \{r_1, r_3\}$). Similar to maximal successful sub queries, (McSherry, 2004) proposes the computation of *minimal failing sub queries* (MFS). A minimal failing sub query is, for example, MFS_1: *size = small* (r_2). This sub query correlates with diagnosis $d_1 = \{r_2\}$. If a minimal failing sub query or diagnosis is shown to the customer, the customer knows, that *all* requirements of this sub query or diagnosis need to be adapted or deleted. Compared to the minimal failing sub queries where all requirements have to be adapted, only one requirement of each conflict set has to be adapted.

This thesis focuses on the identification of minimal conflict sets and minimal diagnoses, which are the result of a *customer requirements diagnosis problem*. A formal definition of a customer requirements diagnosis problem is given in Definition 3 and it is compatible with (Felfernig et al., 2004).

Definition 3 *A **Customer Requirements Diagnosis Problem** is defined as a tuple (C_{KB}, C_R) where C_R is the set of customer requirements and C_{KB} represents the configuration knowledge base or products which*

can be either a product table or a set of product constraints.

Based on this Customer Requirements Diagnosis Problem, minimal conflict sets can be defined (based on the work of (Reiter, 1987) and (Felfernig et al., 2004)). These minimal conflict sets are defined as:

Definition 4 *A **conflict set** for a customer requirement diagnosis problem (C_{KB}, C_R) is a set $CS \subseteq C_R$, s.t., $C_{KB} \cup CS$ is inconsistent. CS is* minimal iff *there does not exist a conflict set $CS' \subset CS$ s.t. $C_{KB} \cup CS'$ is inconsistent.*

In other words, a conflict set is a subset of the given requirements of the customer in a way that none of the items in product assortment or no configuration emerging from the knowledge base satisfies all constraints in *CS*. Customers can be supported in the consistency management by minimal conflict sets. In each step of a continuous interaction the customers are confronted with one minimal conflict set. The customers have to adapt or delete one of the elements of the conflict set until they find a consistent situation. Another possibility is to suggest diagnoses to the customers. The definition of a Customer Requirements Diagnosis is given in Definition 5 and it is compatible with the work of (Reiter, 1987) and (Felfernig et al., 2004).

Definition 5 *A **diagnosis** for a customer requirement diagnosis problem (C_{KB}, C_R) is a set $D \subseteq C_R$, s.t., $C_{KB} \cup (C_R - D)$ is consistent. D is* minimal iff *there does not exist a diagnosis $D' \subset D$ s.t. $C_{KB} \cup (C_R - D')$ is consistent.*

A diagnosis is a set of requirements that need to be adapted or deleted in order to restore consistency. Furthermore, it is the complement of a maximal successful sub-query. In general, the problem to find a maximal succeeding sub-query or a minimal diagnosis (relaxation) has been shown to be NP-hard (Godfrey, 1997). Considering the hard real-time requirement of interactive constraint-based systems (see Section 1.2), even small-sized problems soon become computationally intractable. Therefore, this thesis introduces algorithms that use heuristics in order to achieve an acceptable run time performance.

Aiding the Customer in Consistency Management

The consistency management techniques that emerge from the field of model-based diagnosis can be used to aid customers in interactive systems. Figure 2.6 and Figure 2.7 exemplify an interaction between a customer and a constraint-based system. In the step (1), the customer specifies his requirements. If these requirements are over-constrained, then the system can not directly derive a solution to the stated problem. In this case (2), two different techniques can be applied to aid the customer in consistency management. The first option is to present one minimal conflict set to the customer in each interaction step. Through several interactions the customer chooses which requirements should be adapted. This is continued until the consistency has been restored (see Figure 2.6).

A second possibility is to aid the customer is to use repair actions (see Figure 2.7). A repair action contains a set of adaptations, that - if accepted - leads to at least one product or configuration. Such a repair action can be calculated on the basis of a diagnosis. A minimal diagnosis incorporates only those

requirements that need to be changed by the customer in order to restore consistency. One possibility would be to suggest these requirements (repairs) directly to the customer so that he can adapt them. The big drawback of this approach (to present diagnoses) is that it may result in another over-constrained situation. This drawback can be avoided by identifying possible assignments of all requirements of one minimal diagnosis (Felfernig et al., 2009c). By accepting these values the consistency can be restored. The set of adaptations is denoted as *repair action*. At least one repair action can be identified for each diagnosis, but in general numerous repair actions are available. If there are more repair actions available for one diagnosis, the most suitable approach is to suggest the ones which are most similar to the original requirements. After the customer has restored the consistency (either by using the minimal conflict technique presented in Figure 2.6 or by using repair actions presented in Figure 2.7) the system can present the result (3). Note, that there exist also other possibilities to ensure the derivation of an item. For example, if in the requirement specification phase (1) only values can be selected, that lead to at least one product.

2.4. Summary

This chapter gives an overview of recommender and configuration systems. The goal of these systems is, to take the requirements specified by the customer and to find a respective recommendation or configuration. If the system fails to find a solution, a procedure to aid the customer in restoring consistency needs to be invoked. Two different techniques can be used for the interaction during an inconsistent situation. The first one uses minimal conflict sets which are resolved by the customer in an interactive way. The other approach is to provide a list of repair actions to the customer.

This work focuses on recommender and configuration systems. The main difference between them is, that the former operates on the product assortment with explicit instantiations of the alternatives, whereas the latter operates on a product model, which describes the properties of all allowed instances. In the following chapters different algorithms are presented that can be used in constraint-based systems to aid customers in consistency management.

Figure 2.6.: Interaction with a constraint-based system using minimal conflict sets to restore the consistency. *Step 1:* the customer specifies the requirements. *Step 2:* the customer adapts one faulty requirement at a time. *Step 3:* the result is presented.

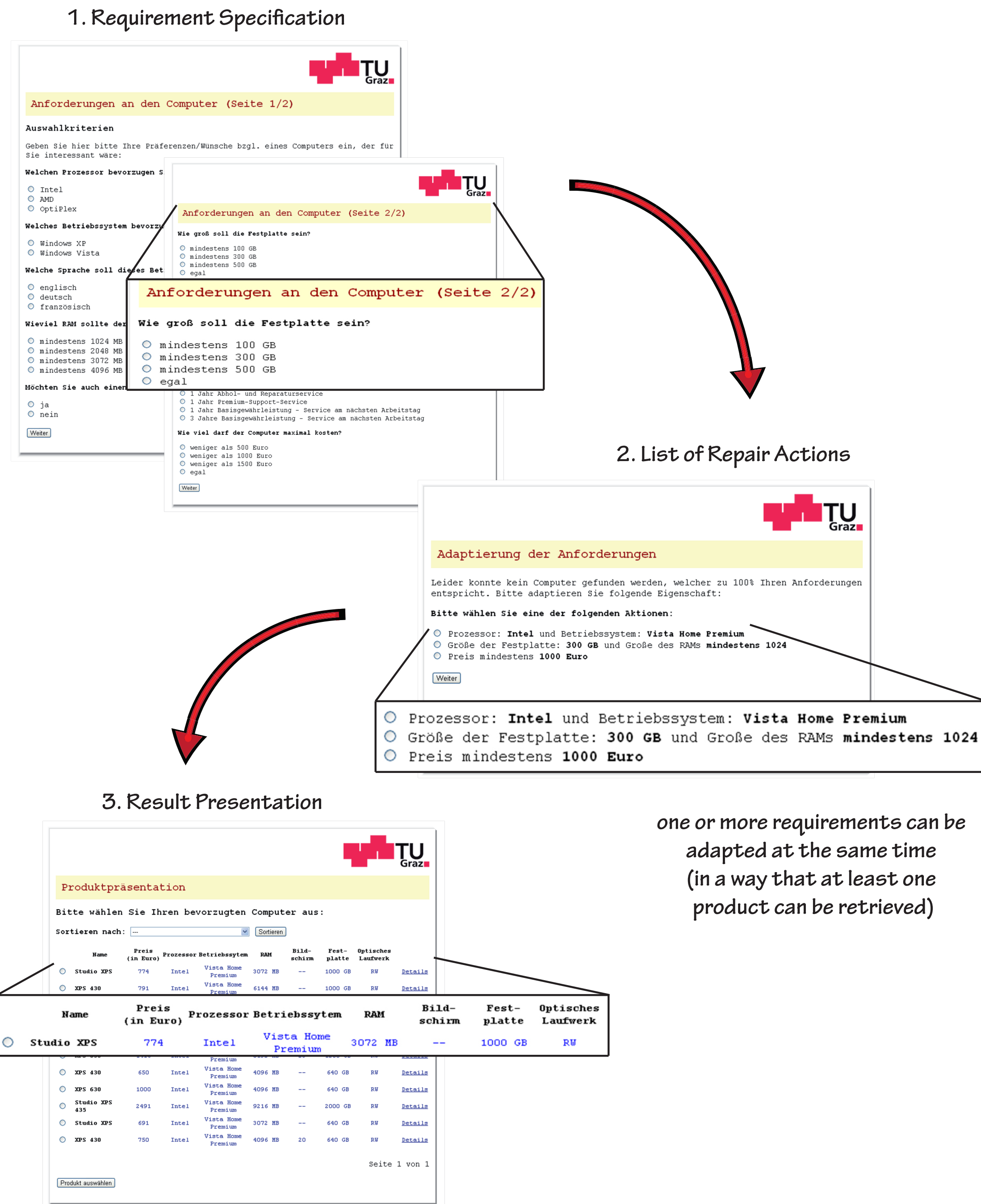


Figure 2.7.: Interaction with a constraint-based system using repair actions to restore the consistency. *Step 1:* the customer specifies the requirements. *Step 2:* the customer selects one repair action. *Step 3:* the result is presented.

Chapter 3

Consistency Management in Recommender Systems

Parts of the contents of this chapter have been published in (Schubert, 2009; Schubert et al., 2009; Felfernig et al., 2009c,b) (Schubert et al., 2010; Schubert and Felfernig, 2011; Schubert et al., 2011).

This chapter introduces different algorithms that can be applied to support the customer in consistency management in knowledge-based recommendation scenarios. Knowledge-based recommenders guide customers to identify interesting items from large and potentially complex assortments. Although there are different methods for eliciting customer requirements as well as for finding and ranking products, all systems commonly treat - at least in the beginning - some or all customer requirements as constraints products have to satisfy (McSherry, 2004). During the process of preference construction, customers incrementally define and revise their requirements. Situations may occur where none of the product items completely fulfils the current set of requirements. Such situations are called the *no solution could be found dilemma* (Pu and Chen, 2008). If such a situation occurs, there are different strategies for the system to deal with it in order to restore the consistency.

One strategy to deal with the *no solution could be found dilemma* is to only notify the customer of the fact. This is not customer-friendly, because the customer can not reach her goal and thus gets emotionally frustrated by the interaction. For this reason, there exist a need for intelligent techniques to aid customers. Another strategy is to propose a small (ideally minimal) set of requirements that have to be changed or given up in order to find a recommendation. The identification of such minimal sets relies heavily on the identification of minimal conflict sets (Reiter, 1987). Existing conflict detection algorithms (like *QuickXplain* introduced by (Junker, 2004)) can be applied, but they do not exploit the basic structural properties of constraint-based recommendation problems. In order to improve the run time efficiency the following algorithms are introduced in this Chapter: *GraphXplain*, *FastXplain*, *Boosted FastXplain* (BFX) and *Personalized FastXplain* (PFX). All these algorithms aim to exploit the structural properties of knowledge-based recommendation problems and improve the run time behaviour for identifying minimal conflict sets. Such algorithms are extremely useful for interactive recommendation sessions. The computed minimal conflicts can be instantly presented to the customer who decides which requirements should remain the

same and which requirements should be changed (interactive repair scenario) (Schubert, 2009).

Another strategy to handle inconsistent situations is to search for product items that fulfil as many individual requirements (constraints) as possible. The aim is to identify a *maximal successfully sub-query* which can be denoted as the complement of a diagnosis. (McSherry, 2004) proposed an algorithm that incrementally searches for minimal exclusion sets based on results in the field of *cooperative query answering* (Godfrey, 1997). The complement of such a minimal exclusion set is a maximally successful sub-query (Jannach, 2008). (O'Sullivan et al., 2007) introduced an approach to identify *representative explanations* which makes the identification of acceptable diagnoses for the customer easier. In a representative set of explanations, each specified customer requirement is included in at least one explanation. The aim of representativeness (diversity) ensures that the offered set of explanations does not get too large but still has a lot of variety in it. This is accomplished by the fact that in the worst-case, the number of representative explanations scales linearly with the number of customer requirements (O'Sullivan et al., 2007). (O'Callaghan et al., 2005) introduced the approach *CorrectiveRelax*, which aims to calculate *corrective explanations* to aid customers in their consistency management (see Section 3.8 for a detailed example).

In general, the problem to find a maximal succeeding sub-query or a minimal diagnosis (relaxation) has been shown to be NP-hard (Godfrey, 1997). If the hard real-time requirement of interactive recommender systems (see Section 1.2) are considered, even small-sized problems become soon intractable. Therefore, different techniques to use better heuristics with acceptable run times were developed.

This chapter introduces techniques and algorithms to aid customers in restoring the consistency between requirements and the recommendation knowledge base. All approaches rely on detailed knowledge about the underlying products and their properties as well as explicit knowledge about the customers requirements.

The remainder of this chapter is organized as follows: Section 3.1 introduces an example from the domain of digital cameras that is used throughout the chapter to illustrate the different approaches. Sections 3.2-3.5 introduce approaches that aim to support customers with the identification of minimal conflict sets. In Sections 3.6 and 3.7 algorithms for identifying minimal diagnoses as well as their respective repair sets are illustrated. All these technologies remedy the *no solution could be found dilemma* (Pu and Chen, 2008) and thus help the customer with consistency management. Section 3.8 presents an evaluation which addresses the run time performance as well as the acceptance probability (precision). This evaluation compares the introduced algorithms with state-of-the-art approaches (*QuickXplain* (Junker, 2004), *CorrectiveRelax* (O'Callaghan et al., 2005) and *MinRelax* (Jannach, 2008)). Section 3.9 discusses related work. The chapter concludes with a decision guide for selecting the algorithms (see Section 3.10).

3.1. Example: Recommending Digital Cameras

A simplified example from the domain of compact camera sales is introduced in this section. The example will serve as a representative problem throughout this chapter to explain the principles of the different algorithms. It describes a *recommendation task* (see Definition 6 according to the work of (Felfernig et al., 2005)), which is defined as a Constraint Satisfaction Problem (CSP (Tsang, 1993)).

Definition 6 *A recommendation task can be defined as a Constraint Satisfaction Problem (V_C, V_P, $C_C \cup C_F \cup R \cup C_P$), where V_C is a set of variables representing possible customer requirements and V_P is a set of variables describing product properties. Moreover, C_P is a set of constraints describing available product instances, R is a set of constraints describing customer requirements and C_F is a set of constraints (filter conditions) describing the relationship between customer requirements and available products. Finally, C_C is a set of concrete customer requirements.*

The product assortment of the working example consists of 15 digital cameras. These items $P = \{p_1, p_2, ..., p_{15}\}$ are stored in the product table (see Table 3.1). The table also specifies attributes $V_P = \{v_1, v_2, ..., v_9\}$ for each product where v_1 : *mpix* specifies the image sensor size (in Megapixel); v_2 : *display* enumerates backside diagonal display size in inches; v_3 : *opt-zoom* describes the optical zoom factor; v_4 : *sound* specifies whether the camera can record audio; v_5 : *waterproof* specifies whether the camera is waterproof or not; v_6 : *movies* indicates that the camera is able to take movies; v_7 : *colour* indicates the main colour of the camera; the v_8 : *weight* is given in pounds and v_9 : *price* specifies the purchase price of the camera in Euro. Customers may specify all or just some of these attributes. In the example, the customer only specified her preferences on some of the attributes. The preferences of the customer are shown in Section 3.1.1. These preferences are interpreted as constraints on the product in constraint-based recommender systems.

Table 3.1.: Example product assortment of digital cameras $P = \{p_1, p_2, ..., p_{15}\}$. For each camera the *id*, the mega pixels (*mpix*), the optical zoom (*opt-zoom*), the ability to recording *sound*, the ability to be *waterproof*, the ability to record *movies*, the *colour*, the *weight* and the *price* are specified.

id	mpix	display	opt-zoom	sound	waterproof	movies	colour	weight	price
p_1	12.0	2.9	6.6x	yes	yes	yes	red	1.4	229
p_2	8.3	4.0	8.0x	yes	no	no	pink	2.0	155
p_3	9.5	3.7	3.2x	no	yes	no	blue	2.3	207
p_4	7.5	3.5	5.0x	no	no	no	silver	1.2	89
p_5	13.0	2.7	5.5x	yes	no	yes	silver	4.7	270
p_6	12.0	2.8	5.8x	no	yes	yes	blue	0.7	249
p_7	5.5	3.2	12.0x	yes	no	no	green	0.63	145
p_8	9.0	3.7	3.0x	no	no	no	black	2.95	79
p_9	7.5	2.8	5.8x	yes	no	no	red	1.7	99
p_{10}	8.0	4.0	3.7x	no	yes	yes	green	2.1	221
p_{11}	11.0	3.5	6.2x	yes	yes	no	black	1.3	240
p_{12}	7.5	3.1	3.5x	yes	no	yes	yellow	0.8	329
p_{13}	12.0	2.5	7.0x	no	yes	yes	pink	1.9	179
p_{14}	9.3	3.9	4.5x	yes	yes	no	black	2.3	299
p_{15}	15.0	2.7	3.8x	no	yes	yes	silver	1.2	199

3.1.1. Requirements of the Customer

Let us assume that the following requirements are specified by the current customer: R = { r_1*: mpix >* *10.0,* r_2*: display* $\geq$ *3.0,* r_3*: opt-zoom* $\geq$ *4x,* r_4*: waterproof = yes,* r_5*: movies = yes,* r_6*: price < 150* }. Based on the recommendation task (see Definition 7) a *recommendation* can be defined according to the work of (Felfernig et al., 2005; Felfernig and Burke, 2008) as follows:

Definition 7 *A* ***recommendation*** *for a recommendation task is an assignment of the variables in* V_C *and* V_P*. A recommendation is denoted as consistent iff each variable in* V_C*,* V_P *has an assigned value and this assignment is consistent with* $C_C \cup C_F \cup R \cup C_P$*.*

The feasibility or consistency of the customer requirements can easily be checked by a relational query $\sigma_{[R]}P$ where $\sigma_{[R]}$ represents the selection criteria of the query on P. A corresponding SQL query would look like this: $\sigma_{[R]}P$ *= SELECT * FROM P WHERE P.mpix > 10.0 AND P.display* $\geq$ *3.0 AND P.opt-zoom* $\geq$ *4x AND P.waterproof = yes AND P.movies = yes AND P.price < 150*. In the working example, $\sigma_{[mpix>10.0]}P$ would result in $\{p_1, p_5, p_6, p_{11}, p_{13}, p_{15}\}$. For the current customer requirements R the query results in an empty set ($\sigma_{[r_1, r_2, r_3, r_4, r_5, r_6]}P = \emptyset$). This means that no solution could be found for these requirements. In such situations, customers are in need of repair actions to help them restore consistency between their request R and the underlying product assortment P.

3.1.2. Intermediate Representation

In situations where no element in the catalogue satisfies all customer requirements, the system can support the customer with minimal relaxations (Jannach, 2008), maximal succeeding sub queries (McSherry, 2004) or repair actions (Felfernig et al., 2009c). (Jannach, 2008) introduced a data representation (see Table 3.2). This representation stores a **1** if an attribute of the product satisfies the specification of the customer. If an attribute does not satisfy the specification of the customer, the value **0** is stored in the data structure. Based on this data structure a so called product-specific relaxation can be retrieved for each product. This product-specific relaxation (PSX) for product p_1 is $PSX(R, p_1) = \{r_2, r_6\}$, for product p_2 it is $PSX(R, p_2) = \{r_1, r_4, r_5, r_6\}$ and so forth (see Table 3.1). The drawback of this approach is the amount of diagnoses. The amount of diagnoses equals the amount of products, which is usually a lot. Additionally, most of these diagnoses are not minimal. Minimal diagnoses are especially relevant as they propose a *minimal* set of changes of the customers original requirements.

(Jannach, 2008) introduced an approach (*MinRelax*) to identify a minimal query relaxation for situations where the given set of customer requirements is inconsistent with the set of available products (similar to the situation described in Section 3.1.1). The main idea of this algorithm is the following: For each product p_i, the algorithm identifies the product-specific relaxation (PSX). Iterating over all possible relaxations, the algorithm checks if the current $PSX(R, p_i)$ is a superset of an already found relaxation. If the $PSX(R, p_i)$ is not a super set, it is stored and all relaxations that are super sets of $PSX(R, p_i)$ are removed. When all minimal relaxations are retrieved, the most promising for the customer is selected via a cost function.

Table 3.2.: Intermediate representation of the customer requirements R = $\{r_1 : mpix > 10.0, r_2 : display \geq 3.0, r_3 : opt-zoom > 4.0, r_4 : waterproof = yes, r_5 : movies = yes, r_6 : price < 150\}$ and the product assortment of digital cameras $P = \{p_1, p_2, ..., p_{15}\}$

id	r_1	r_2	r_3	r_4	r_5	r_6
p_1	1	0	1	1	1	0
p_2	0	1	1	0	0	0
p_3	0	1	0	1	0	0
p_4	0	1	1	0	0	1
p_5	1	0	1	0	1	0
p_6	1	0	1	1	1	0
p_7	0	1	1	0	0	1
p_8	0	1	0	0	0	1
p_9	0	0	1	0	0	1
p_{10}	0	1	1	1	1	0
p_{11}	1	1	1	1	0	0
p_{12}	0	1	0	0	1	0
p_{13}	1	0	1	1	1	0
p_{14}	0	1	1	1	0	0
p_{15}	1	0	0	1	1	0

Each product-specific relaxation (PSX) can be seen as a diagnosis which - similar to a relaxation - need not to be minimal. This implies, that all diagnoses can be directly calculated from the *intermediate representation*. This determination of diagnoses is used in the algorithms *FastXplain* (see Section 3.3), *BFX* (see Section 3.4 and *PFX* (see Section 3.5). In addition to this method of retrieving diagnoses, the algorithms use a consistency check based on a set of requirements R and a set of products P. Such a consistency check evaluates to *true* if any product $p_i \in P$ satisfies all requirements in R. If no product can be found satisfying all requirements of R, the consistency check fails.

In summary, the introduced intermediate data representation is convenient for identifying diagnoses as well as for performing consistency checks. However, the table must be recalculated every time the requirements change. The computational power needed for this recalculation depends on how many requirements are specified.

3.2. Algorithm: GraphXplain

This section describes the algorithm *GraphXplain* which was published in (Schubert et al., 2009). The key idea behind the *GraphXplain* algorithm is to analyse the network of customer requirements and products. While analysing the structure of that network, more knowledge can be gained about which requirements should change in order to retrieve a product. The intermediate data structure described in Section 3.1.2 can be understood as an adjacency matrix. This adjacency matrix relates the set of products P to the customer requirements R. This structure can be interpreted as a two-mode network (also referred as bipartite network). Two-mode networks are widely used in the field of social network analysis (Wasserman and Faust,

1994). Such a network can either be represented as a matrix (see Table 3.2) or in a graph (see Figure 3.1).

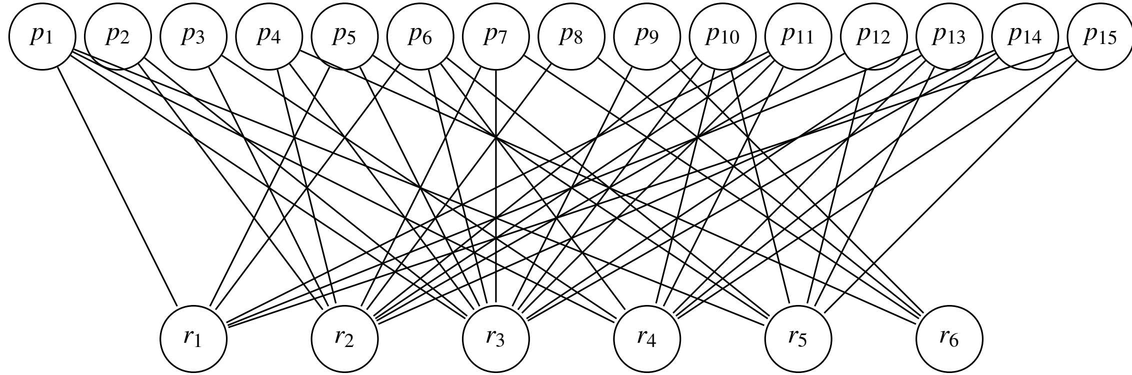


Figure 3.1.: Two-mode network of the customer requirements $R = \{r_1, r_2, ..., r_6\}$ and the digital cameras $P = \{p_1, p_2, ..., p_{15}\}$. An edge in this network is drawn, if the product satisfies the requirement.

In a two-mode network there are only edges between the two groups of items. In the working example, the set of products P is one group and the set of requirements R is the other one. One way to analyse such a two-mode network is to project (or fold) (Wasserman and Faust, 1994) it into a one-mode network of one of the two possible amplitudes. In order to analyse the structure of requirements, the adjacency matrix M is projected to a network of requirements $N_R* = M^T M$. Figure 3.2 shows this network for the working example. The nodes in this network represent the requirements $R = \{r_1, r_2, ..., r_6\}$. An edge between two nodes is drawn, if they share at least one product in the two-mode network. The products connecting edges are:

- r_1-r_2: p_{11}
- r_1-r_3: $p_1, p_5, p_6, p_{11}, p_{13}$
- r_1-r_4: $p_1, p_6, p_{11}, p_{13}, p_{15}$
- r_1-r_5: $p_1, p_5, p_6, p_{13}, p_{15}$
- r_2-r_3: $p_2, p_4, p_7, p_{10}, p_{11}, p_{14}$
- r_2-r_4: $p_3, p_{10}, p_{11}, p_{14}$
- r_2-r_5: p_{10}
- r_2-r_6: p_4, p_7, p_8
- r_3-r_4: $p_1, p_6, p_{10}, p_{11}, p_{13}, p_{14}$
- r_3-r_5: $p_1, p_5, p_6, p_{10}, p_{13}$
- r_3-r_6: p_4, p_7, p_9
- r_4-r_5: $p_1, p_6, p_{10}, p_{13}, p_{15}$

Taking a closer look at the edge $\{r_1$-$r_2\}$, the algorithm knows from the one-mode network that the product p_{11} satisfies both requirements. If only these two requirements are considered to be satisfied the product p_{11} can be recommended. Another example is the edge $\{r_1, r_3\}$ which results in the recommendation of

$\{p_1, p_5, p_6, p_{11}, p_{13}\}$. In comparison to this, there is no edge between r_1 and r_6, because no product satisfies both requirements. Therefore, one of these two requirements needs to be adapted or relaxed in order to retrieve a solution that satisfies all requirements. The goal of the *GraphXplain* algorithm is to identify these requirements that need to be adapted (or relaxed) in order to find a solution that satisfies all requirements. In terms of the graphical representation, this means that the goal is achieved, when a fully connected graph is given, i.e. all edges sharing at least one common product.

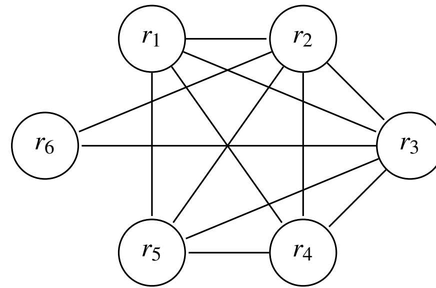


Figure 3.2.: One-mode network of the customer requirements $R = \{r_1, r_2, ..., r_6\}$ based on the two-mode network of Figure 3.1

Identification of Minimal Conflict Sets

When identifying minimal conflict sets by using a graph, two cases have to be distinguished. First, minimal conflict sets with the cardinality $c = 1$ have to be calculated, and second, calculating minimal conflict sets with a higher cardinality ($c \geq 2$). For the calculation of minimal conflict sets of the cardinality $c = 1$, the algorithm needs to analyse the two-mode network. If there exists at least one requirement that is not satisfied by any product, this is a minimal conflict set. This requirement is then a single node in the graph, that is not connected to any other requirement. In other words, this requirement has to be changed to restore consistency. Note that possibly more than just one requirement needs to be adapted. If this is the case, the conflict set with cardinality one is definitely part of the set of requirements that have to be adapted. Note that for the example introduced in Section 3.1, there is no any minimal conflict set with the cardinality one.

For identifying minimal conflict sets with a higher cardinality we first look at a simple example with the cardinality $c = 2$. In order to identify conflict sets containing two elements, the algorithm simply looks for missing edges. Such a missing edge means that there exists no product that satisfies these two requirements and thus at least one of them has to be altered in order to restore consistency. When applying this step to the example, it can be seen that there are 3 edges missing - one between r_1 and r_6 another one between r_4 and r_6 and the last one between r_5 and r_6. Therefore, the algorithm has found three minimal conflict sets: $\{\{r_1, r_6\}, \{r_4, r_6\}, \{r_5, r_6\}\}$. Between all other nodes there is at least one product that the requirements have in common.

This idea can be expanded to the cardinality of three. Therefore, the edge between two nodes can be seen as a fully connected sub-graph of these nodes. For the identification of minimal conflict sets the algorithm analyses all fully connected sub-graphs with 3 nodes. If the edges of these fully connected sub-graphs do not share any product, there is no product that satisfies this subset of requirements and therefore, a conflict

set has been found. The *GraphXplain* algorithm retrieves the conflict sets of the lowest cardinality first. For the algorithm it is important, that it starts with the cardinality of 1 and increases the cardinality. Note that the cardinality of the minimal conflict set is the same as the number of nodes in the sub-graph. For each conflict set the algorithm checks whether a subset has already been found that is a minimal conflict set. If this is the case, there is no need to store the conflict set, because it is not minimal.

Applying the next step (identifying minimal conflict sets with cardinality 3) to the working example introduced in Section 3.1, the sub-graph $\{r_1 - r_2 - r_3\}$ is considered. It can be seen that all edges in this sub-graph (which are: $\{r_1 - r_2\}$, $\{r_2 - r_3\}$ and $\{r_3 - r_1\}$) have at least one product in common. This is the product $\{p_{11}\}$. Therefore, the set $\{r_1, r_2, r_3\}$ is not a conflict set. Considering all fully connected sub-graphs with three nodes, the algorithm comes across the sub-graph $\{r_1 - r_2 - r_5\}$. For all edges of this sub-graph ($\{r_1 - r_2\}$, $\{r_2 - r_5\}$ and $\{r_5 - r_1\}$) the algorithm cannot find any product that is shared by all edges. Therefore, all nodes of this sub-graph are part of a conflict set. Now it is interesting, whether this conflict set ($\{r_1, r_2, r_5\}$) is minimal. For this reason, the set is compared with all other minimal conflict sets that have been retrieved so far. As no one of the retrieved conflict sets is ruled out to be a superset of the current conflict set ($\{r_1, r_2, r_5\}$), it can be ensured, that it is a minimal one.

In the next step the algorithm iterates over all fully connected sub-graphs with 4 nodes (requirements). The sub-graph is $\{r_1 - r_2 - r_3 - r_4\}$. The edges of this sub-graph share the product p_{11}. Therefore, it is not a conflict set. The sub-graph $\{r_1 - r_2 - r_3 - r_5\}$ does not share any product and is therefore a conflict set. Nevertheless, it can be identified not to be minimal, because $\{r_1, r_2, r_5\}$ was already identified as a minimal conflict set. Similarly, the sub-graph $\{r_1 - r_2 - r_4 - r_5\}$ as well as the only fully connected sub-graph containing 5 nodes (sub-graph $\{r_1 - r_2 - r_3 - r_4 - r_5\}$) are conflict sets, but not minimal ones. Another possibility is to eliminate all super sets of already identified conflict sets before continuing. Depending on implementation, this can be faster. Finally, the algorithm has found all minimal conflict sets of the example introduced in Section 3.1 which are: $MCS = \{\{r_1, r_6\}, \{r_4, r_6\}, \{r_5, r_6\}, \{r_1, r_2, r_5\}\}$.

Algorithm 1 GraphXplain (M)

```
{Input: M - adjacency matrix of constraints and items}
MCS ← retrieveSingleNodes(M)
G ← M^T M
MCS ← MCS ∪ retrieveMissingEdges(G)
for all subgraphs do
  SG ← subgraphs.next
  if notHasCommonEdge(SG) then
    set ← nodes(SG)
    if isNoSuperSet(set, MCS) then
      MCS ← MCS ∪ set
    end if
  end if
end for
return MCS
{Output: return MCS including all minimal conflict sets}
```

Description of the *GraphXplain* Algorithm

The idea behind the algorithm *GraphXplain* (see also Algorithm 1) was just described. Here is a more formal explanation of the algorithm. As shown in Algorithm 1, the input to the algorithm is an adjacency matrix **M**. This matrix is similar to the one in Table 3.2. Moreover, in this matrix there is a **1** if the product satisfies the requirement and a **0** otherwise. This matrix can be seen as a two-mode network from which the algorithm retrieves all nodes that are not satisfied by any product (**retrieveSingleNodes(M)**). If any of these single nodes exist, these are minimal conflict sets and thus the algorithm stores them in the variable **MCS**. In the next step the algorithm projects the two-mode adjacency matrix into a requirement-mode graph **G**. This is done by calculating $M^T M$. Note that the matrix references the requirements in the columns and the products in the rows. If this graph **G** is not fully connected, the algorithm retrieves the missing edges (**retrieveMissingEdges(G)**) and stores the minimal conflict sets in the variable **MCS**.

Furthermore, the algorithm iterates over all fully connected sub-graphs. The method **notHasCommonEdge(SG)** checks if all edges within this graph have at least one such product in common (**notHasCommonEdge(SG)**). If there exists at least one product in all edges, then the set of requirements (nodes) is no conflict set and the algorithm can continue with the next sub-graph. If the edges do not share any product, then a conflict set has been found. The goal of the algorithm is to identify only minimal conflict sets and the algorithm needs to check if it has found a subset of the current set that is already a minimal conflict set. If the set is not a super set of any minimal conflict set (**isNoSuperSet(set, MCS)**), then the algorithm can add it to the set of minimal conflict sets **MCS**. If there are no more fully connected sub-graphs, the algorithm terminates.

3.3. Algorithm: FastXplain

This section presents the algorithm *FastXplain* which was published in (Schubert et al., 2010). The goal of the algorithm is to identify minimal conflict sets in constraint-based recommender systems in a fast way. In order to achieve this goal, the algorithm evaluates the situation (requirements of the customer as well as products) and takes the structural properties into account. This structural property is a data table similar to the one in Table 3.2. Based on this data representation all diagnoses can be directly extracted: for every column (product) take every requirement where the product does not satisfy the requirement (a zero is in the table). This set of requirements (diagnosis) provides a relaxation to the query containing all requirements of the customer (see also Section 3.1.2 and (Jannach, 2008)).

Identification of Minimal Conflict Sets

In order to support a customer in interactive settings, *FastXplain* calculates minimal conflict sets based on minimal diagnoses. As already described in Section 2.3 a diagnosis is a set of requirements that need to be adapted or deleted in order to restore the consistency (meaning that at least one product can be recommended). A diagnosis is based on a set of conflicts. Each conflict between the customer requirements and the product assortment has to be resolved to achieve a consistent state. Therefore a diagnosis contains one element of each conflict set. Based on this (Reiter, 1987) introduced the hitting set directed acyclic graph to determine all minimal diagnoses based on minimal conflict sets (also called hitting sets). *FastXplain*

takes the same property between diagnoses and conflict sets to identify all minimal conflict sets based on minimal diagnoses.

For identifying the minimal conflict sets, *FastXplain* iterates over all products and retrieves the diagnoses of the lowest cardinality (lowest number of requirements in the diagnosis). In the working example from Section 3.1 there are three diagnoses with the cardinality 2, i.e. $\{r_2, r_6\}$, $\{r_1, r_6\}$ and $\{r_5, r_6\}$. The diagnosis $\{r_2, r_6\}$ is retrieved first. The algorithm can start building a directed acyclic graph (similar to the concepts in (Reiter, 1987)). This original graph is adapted in a way that diagnoses are added instead of conflict sets as in the original. The root node of the graph represents the current situation *(R, P)* with R being a set of requirements to relax and P the set of products obtainable by this relaxation. The edges represent single requirements that need to be relaxed. All edges leaving one particular node are the elements of one diagnosis. For example, in Figure 3.3 (which shows the directed acyclic graph of the working example), the requirements leaving the root node (r_2 and r_6) are the first diagnosis.

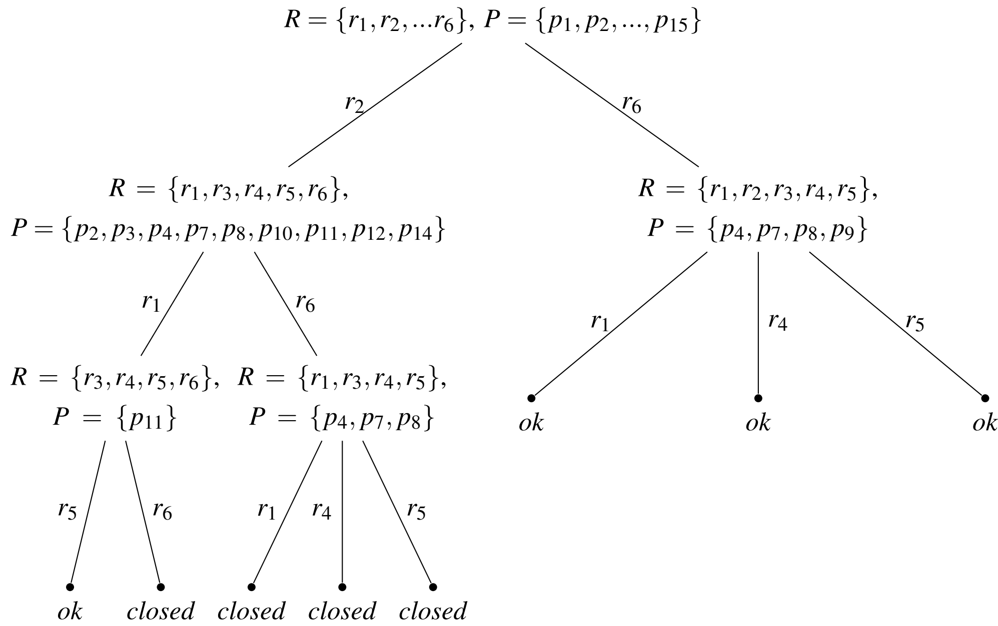


Figure 3.3.: Directed acyclic graph built from diagnoses to identify minimal conflict sets. The graph is constructed by the algorithm *FastXplain* from the example of Section 3.1

The root node of the graph holds all customer requirements $R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ and all products $P = \{p_1, p_2, ..., p_{15}\}$. Based on this input the algorithm *FastXplain* is called and retrieves the first diagnosis ($d_1 = \{r_2, r_6\}$) from the intermediate data table (see Table 3.2). Each element of the diagnosis d_1 is added as edge to the root node with the label of the requirement. After adding all labelled edges, the algorithm calculates the product set for the children. Let us take a closer look at the edge $\{r_2\}$. For analysing the situation of the child at the end of the edge, the algorithm excludes the requirement r_2 from the set of requirements R. In addition, the algorithm needs to adapt the set of products. This is done by deleting all products from the assortment which do not satisfy the requirement r_2. The reason for this elimination is that r_2 is part of the conflict set and in order to find another element of the conflict set the algorithm searches in these products that satisfy r_2. For the working example the set of remaining products is $P =$

$\{p_2, p_3, p_4, p_7, p_8, p_{10}, p_{11}, p_{12}, p_{14}\}$. Similarly, the algorithm identifies requirements and products for the second branch in the graph.

In the next iteration the algorithm is applied to the leafs, first using the smaller set of requirements and products of the left leaf (following the left edge $\{r_2\}$ in Figure 3.3). The diagnosis with the lowest cardinality of the left node ($R \setminus \{r_2\} = \{r_1, r_3, r_4, r_5, r_6\}$ and $P_{[\neg r_2]} = \{p_2, p_3, p_4, p_7, p_8, p_{10}, p_{11}, p_{12}, p_{14}\}$) is $d_2 = \{r_1, r_6\}$. If all products have been eliminated from the current product set which do not satisfy r_1, there is still a product (p_{11}) left in the set. There are also products left for the requirement r_6 (i.e. p_4, p_7, p_8). As there are still products left, the algorithm has not found a minimal conflict set yet. This is based on the fact, that a conflict set is a set of requirements where the product query gives an empty result (see Definition 4). Therefore, the algorithm needs to expand the tree further in a breadth first manner. The next node the algorithm expands is the one with the requirements $R \setminus \{r_6\} = \{r_1, r_2, r_3, r_4, r_5\}$ and the product set $P_{[\neg r_6]} = \{p_4, p_7, p_8, p_9\}$ (see the right branch of the tree in Figure 3.3). The lowest cardinality diagnosis for this set of products is $d_3 = \{r_1, r_4, r_5\}$. Checking the node with the path $\{r_6 - r_1\}$, the algorithm can eliminate from the current set of products all products that do not satisfy the requirement r_1 ($P_{[\neg r_6, \neg r_1]} = \{\}$). This set of products is empty and therefore, the algorithm has found a minimal conflict set, $mcs_1 = \{r_1, r_6\}$. Breadth-first expansion ensures, that all possible sets of cardinality c are evaluated before any sets of cardinality c+1 are. This fact ensures the minimalism property of the conflict sets. To gather all minimal conflict sets, this method is iterated until each leaf has either an empty set of remaining products or the path to the leaf is not minimal (the set of requirements of the path is a superset of another minimal conflict set). Following this procedure for the introduced example (see Section 3.1), the algorithm ends up with the set of minimal conflict sets $MCS = \{\{r_1, r_6\}, \{r_4, r_6\}, \{r_5, r_6\}, \{r_1, r_2, r_5\}\}$.

Description of the *FastXplain* Algorithm

This section dives into a more formal description of *FastXplain* which can be seen in Algorithm 2. The *FastXplain* algorithm was inspired by the hitting set directed acyclic graph (HSDAG) (Reiter, 1987) as well as by (Jannach, 2008), thus we take over the same kind of description. The input values for *FastXplain* are a **root** node for the graph as well as a set of relaxed requirements **R** and a set of obtainable products **P**. Note that it is enough to store only the indexes of the requirements and the products in the sets, because the algorithm operates on the intermediate data representation (see Table 3.2). The **root** node is the reference to the resulting graph which contains the diagnoses as well as the minimal conflict sets. The result of the *FastXplain* algorithm is a directed acyclic graph (tree similar to Figure 3.3) as well as a set of minimal conflict sets stored in the global variable **MCS**. This variable is initialized to an empty set.

During the first step, *FastXplain* retrieves the diagnosis with the lowest cardinality from the intermediate data representation (see Table 3.2). This is done by the function **getMinCardinalityDiagnosis**. This function operates on the current set of requirements **R** and the current set of products **P**, which are provided as parameters. The result is the lowest cardinality diagnosis - the one consisting of the lowest number of requirements. If there are more diagnoses with the same number of requirements, the algorithm takes the first one. For each requirement of the diagnosis **D** a set of products **P'** containing all products that satisfy this requirement is calculated (**reduce(P,r)**) - i.e. all products that do not satisfy the requirement are deleted from the original set of products **P**. Based on this reduced set of products **P'** combined with the remaining set of requirements (**R'=**$R \setminus \{r\}$, where **r** is deleted from **R**) as well as the current requirement **r** as label

Algorithm 2 FastXplain(root, R, P)

```
{Input: root - the root node of the current subtree}
{Input: R - set of user requirements}
{Input: P - set of products}
{Global: MCS - set of all minimal conflict sets}
D ← getMinCardinalityDiagnosis(R, P)
for all requirements r from D do
   P' ← reduce(P, r)
   child ← addChild(r, R \ {r}, P')
   if P' = {} then
      if path(child) ∉ MCS then
         MCS ← path(child)
         child ← ok
         return
      end if
      child ← closed
   end if
   if ∃ cs ∈ MCS : cs ⊆ path(child) then
      child ← closed
   end if
   if child ≠ closed then
      FastXplain(child, R \ {r}, P')
   end if
end for
```

for the edge a node in the graph is created and added to the current root node (**addChild**). This node is stored in the variable **child**. If the set of products **P'** is empty, a conflict set (path from the root node of the graph to the child) has been found. The path to the child is a *minimal conflict set*, if there is no element in **MCS** (set of all minimal conflict sets) which is a subset of the current path. If the current path is a minimal conflict set, it is added to the variable **MCS** and the node is marked as *ok*. If it is not a minimal conflict set, the node is marked as *closed*. If a node is marked (either with *ok* or *closed*), there is no need to further investigate into this node.

If the set of products **P'** is not empty, the algorithm checks if there exists a minimal conflict set **cs** in **MCS**, that is a subset of the current path. If this is the case, the current path cannot be a minimal conflict set any more and thus the algorithm does not need to further investigate into it. As a consequence the status of the node is set to *closed*. In all other cases the tree is constructed further. When calculating all minimal conflicts sets, a tree is created where all paths to the leafs marked with *ok* are minimal conflict sets. Otherwise (for non-minimal conflict sets) the leafs are marked with *closed*.

3.4. Algorithm: Boosted FastXplain (BFX)

This section explains the *BFX (Boosted FastXplain)* algorithm which was published in (Schubert and Felfernig, 2011). This algorithm is an extension of the *FastXplain* algorithm that has been introduced in Section 3.3. The goal of this algorithm is to improve the run time performance of the *FastXplain* algorithm which is important to improve the usability of the system. The *BFX* algorithm uses the same underlying data structure as *FastXplain* (see Table 3.2). The main difference between the two algorithms is that the *BFX* algorithm uses a weighting criteria to identify the sequence of the diagnoses. This effects the appearance of the hitting set directed acyclic graph (HSDAG) (Reiter, 1987).

Identification of Minimal Conflict Sets

The *BFX* algorithm operates on the data structure based on the customer requirements and the products (see Table 3.2) to identify all minimal conflict sets. A simple approach how to identify all diagnoses is introduced in (Jannach, 2008). A more advanced approach is the *FastXplain* algorithm described in Section 3.3 and (Schubert et al., 2010). This algorithm can be improved by introducing a weight to identify the 'best' diagnoses. This weight takes into account how often each requirement is satisfied. Therefore it calculates the sum of each column (see Table 3.3) in a first step. In Table 3.3 (which is an extension of Table 3.2) the last row shows the sum of each column. The requirement r_1 is satisfied by 6 products ($w(r_1) = 6$), requirement r_2 is satisfied by 9 products ($w(r_2) = 9$), and so on. Based on this sum the total weight of each diagnosis can be calculated. This is done by summing up the sum value of each requirement which is part of the diagnosis of this product. Considering the product p_1, the diagnosis that can be retrieved from this product is $\{r_2, r_6\}$. For each element of the diagnosis the values of the requirements r_j are summed up. For the requirement r_2 this value is 9 and for the requirement r_6 it is 4. The sum of these values is 13. This is the weight for the diagnosis $\{r_2, r_6\}$. On a more formal level:

$$weight(d_i) = \sum (1 - v_j) * w(r_j) \tag{3.1}$$

Note that for each requirement that is part of the diagnosis, the value (v_j) is **0** in the table representation. For all other requirements the value (v_j) is **1**. $w(r_j)$ is the number of products that satisfy the requirement r_j. The weights of diagnoses derived from all products are shown in Table 3.3.

After the calculation of all weights, the diagnosis with the lowest weight is selected. The weight of a diagnosis indicates how many products can be recommended when all requirements of this diagnosis are adapted or deleted. For the example this is the diagnosis $d_1 = \{r_1, r_6\}$ with the weight 10 resulting from product p_{10}. Based on this selected diagnosis d_1 the algorithm can build a directed acyclic graph similar to the *hitting set directed acyclic graph* (Reiter, 1987). This graph is built of nodes which hold information about the current set of requirements as well as the current set of products. The edges are labelled with requirements. All outgoing edges are part of one diagnosis and all paths from the root node to the leafs are conflict sets (see also Section 3.3). The graph constructed by the *BFX* algorithm can be seen in Figure 3.4. Similar to the *FastXplain* algorithm the root node consists of the customer requirements $R = \{r_1, r_2, ..., r_6\}$ and all products $P = \{p_1, p_2, ..., p_{15}\}$ from the product table.

Table 3.3.: Intermediate representation for the *BFX* algorithm of the customer requirements R and the product assortment P of digital cameras. In the last row the weight of each requirement is given. The weight ($weight(d_i) = \sum(1 - v_j) * w(r_j)$) of each diagnosis is given in the last column.

id	r_1	r_2	r_3	r_4	r_5	r_6	$weight(d_i)$
p_1	1	⓪	1	1	1	⓪	13
p_2	0	1	1	0	0	0	25
p_3	0	1	0	1	0	0	28
p_4	0	1	1	0	0	1	21
p_5	1	0	1	0	1	0	21
p_6	1	0	1	1	1	0	13
p_7	0	1	1	0	0	1	21
p_8	0	1	0	0	0	1	32
p_9	0	0	1	0	0	1	30
p_{10}	0	1	1	1	1	0	10
p_{11}	1	1	1	1	0	0	11
p_{12}	0	1	0	0	1	0	29
p_{13}	1	0	1	1	1	0	13
p_{14}	0	1	1	1	0	0	17
p_{15}	1	0	0	1	1	0	24
Σ	6	⑨	11	8	7	④	

The first diagnosis retrieved is $d_1 = \{r_1, r_6\}$. For each element of this diagnosis an edge is added to the root node. Afterwards the algorithm calculates the remaining set of requirements and the remaining set of products similar to *FastXplain*. For the edge $\{r_1\}$ the set of remaining products is $P_{[\neg r_1]} = \{p_1, p_5, p_6, p_{11}, p_{13}, p_{15}\}$ and the set of remaining requirements (eliminating r_1 from R) is $R \setminus \{r_1\} = \{r_2, r_3, r_4, r_5, r_6\}$. And for the edge $\{r_6\}$ the set of remaining products is $P_{[\neg r_6]} = \{p_4, p_7, p_8, p_9\}$ and the set of remaining requirements is $R \setminus \{r_6\} = \{r_1, r_2, r_3, r_4, r_5\}$. Continuing with the node at the end of edge $\{r_1\}$ the diagnosis $d_2 = \{r_5, r_6\}$ is the one with the lowest weight which can be derived from the remaining set of products $P_{[\neg r_1]}$. For each requirement of the diagnosis, an edge is added to the current node as well as the remaining set of products. For the edge $\{r_1 - r_5\}$ this set is $P_{[\neg r_1, \neg r_5]} = \{p_1, p_5, p_6, p_{13}, p_{15}\}$ and for the edge $\{r_1 - r_6\}$ the set of products is empty. A minimal conflict set has been found. The node can be marked with *ok* and the algorithm continues in a breath first manner.

For calculating all minimal conflict sets, this method has to be continued until each leaf has either an empty set of remaining products or the path to the leaf is not minimal (the set of constraints of the path is a superset of a minimal conflict set). The full graph is shown in Figure 3.4. From this graph all minimal conflict sets of the example can be easily identified. These minimal conflict sets are: $MCS = \{\{r_1, r_6\}, \{r_4, r_6\}, \{r_5, r_6\}, \{r_1, r_2, r_5\}\}$.

Comparing the graph constructed by the *FastXplain* algorithm (see Figure 3.3) with the one built by the *BFX* algorithm (see Figure 3.4) it can be observed that the second graph constructed by the *BFX* is smaller. The size of the tree is reflected in an improvement of the run time as well as in the memory needed. The improvement of the *BFX* algorithm is based on weights. The weight of a requirement is an indicator of

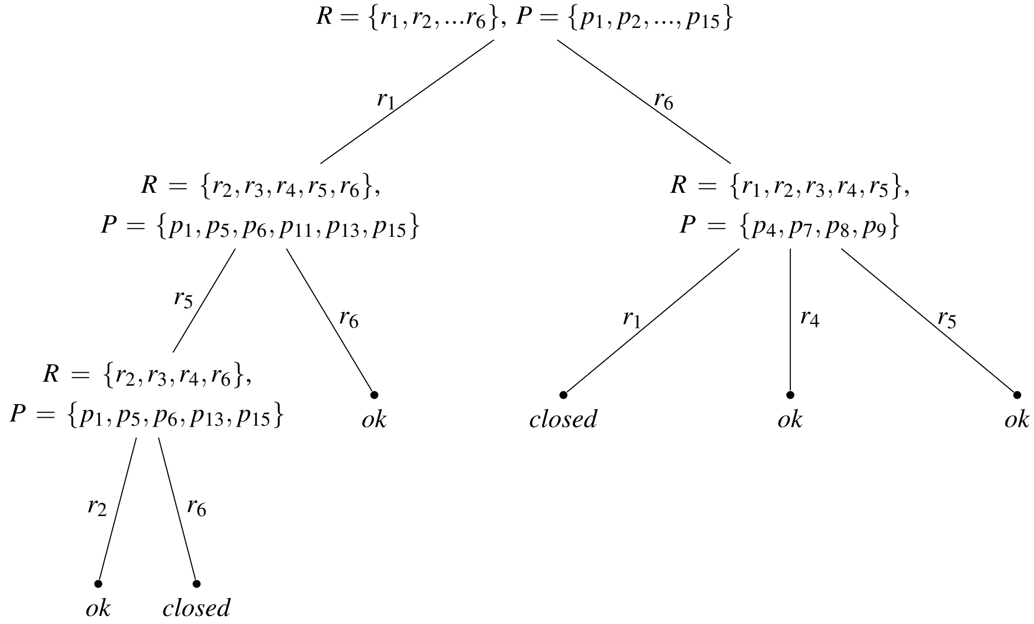


Figure 3.4.: Directed acyclic graph built of diagnoses to identify minimal conflict sets. The graph is constructed by the algorithm *BFX* (Boosted FastXplain) from the example of Section 3.1

how valuable it is to adapt this requirement. A high weight of a requirement indicates that it is satisfied by a lot of products. Therefore, if the customer relaxes a high weighted requirement, only few more products are satisfied. Nevertheless, the aim of the algorithm is to leave the inconsistent state as fast as possible. For this reason, the algorithm is interested in low weighted requirements because they lead to more useful products (for the customer).

Description of the *BFX* Algorithm

After describing the ideas behind the *BFX* algorithm, this section focuses on a more formal description of it (see Algorithm 3). The algorithm was inspired by *FastXplain* (see Section 3.3 and (Schubert et al., 2010)) thus the same level of description is kept. The input values of the *BFX* algorithm are the **root** node for the graph, a set of customer requirements **R** and a set of products **P**. The *root* node is the reference of the resulting graph which holds all information about the diagnoses and the minimal conflict sets. The result of the algorithm are all minimal conflict sets which are stored in the global variable **MCS**. This variable is empty in the beginning. As a side product the algorithm generates a directed acyclic graph that holds all minimal diagnoses. For example, the root node holds the diagnosis $d_1 = \{r_1, r_6\}$. Similarly, the other diagnoses of the nodes of the left side for the graph are: $d_2 = \{r_5, r_6\}$, $d_4 = \{r_2, r_6\}$ and the diagnosis of the node of the right side is $d_3 = \{r_1, r_4, r_5\}$.

In a first step *BFX* retrieves the diagnosis with the lowest weight (**getMinWeightDiagnosis(R,P)**) based on the set of requirements **R** and the set of products **P**. The weight of a diagnosis d_i based on the product p_i is given by $weight(d_i) = \sum(1 - v_j) * w(r_j)$. v_j is the value for each attribute for the product p_i in the data structure (see Table 3.3). And $w(r_j)$ indicates how often the requirement is satisfied (see the sum in the last

Algorithm 3 BFX(root, R, P)

{Input: root - the root node of the current subtree}
{Input: R - set of user requirements}
{Input: P - set of products}
{Global: MCS - set of all minimal conflict sets}
$D \leftarrow getMinWeightDiagnosis(R,P)$
for all *requirements r from D* **do**
  $P' \leftarrow reduce(P,r)$
  $child \leftarrow addChild(r, R \setminus \{r\}, P')$
  if $P' = \{\}$ **then**
    if path(child) $\notin$ MCS **then**
      $MCS \leftarrow path(child)$
      $child \leftarrow ok$
      return
    end if
    $child \leftarrow closed$
  end if
  if $\exists\ cs \in MCS : cs \subseteq path(child)$ **then**
    $child \leftarrow closed$
  end if
  if child $\neq$ *closed* **then**
    $BFX(child, R \setminus \{r\}, P')$
  end if
end for

row of Table 3.3). If there exists more than one diagnosis with the same weight the first one is chosen. For each requirement **r** from the diagnosis the set of products is adapted. This is done by removing all products from the current set **P** which do not satisfy the requirement **r**. The result is stored in the variable **P'**. For the next node the algorithm has to adapt the set of requirements by deleting the requirement **r** from **R**. Finally the algorithm can create a new node at the end of the edge **r** with the label of the edge (**r**), the set of adapted requirements **R' =** $R \setminus \{r\}$ and the set of adapted products **P'**. This node is added to the current root node using the method **addChild**.

If the set of adapted products **P'** gets empty, a conflict set has been found. This conflict set is the path from the root node of the graph to the leaf. If this conflict set is not a superset of any minimal conflict set retrieved so far, then it is a minimal conflict set. In addition, the minimalism of the conflict sets is ensured by a breadth first search of the algorithm. If the algorithm identifies a minimal conflict set, the node can be marked with *ok*. Furthermore, the algorithm closes this node. This means that no further investigation is needed for this leaf. If this path to the node is not an element of **MCS** (set of all minimal conflict sets) yet, then it is added. If the path or a subset of it is already an element of the minimal conflict sets retrieved so far, then there is no need to expand the node any more. This is based on the fact that the conflict set would not be minimal. In this situation the algorithm can mark the node as *closed*.

In all other cases the graph is constructed further in breadth-first manner. If the *BFX* algorithm is not

Table 3.4.: Utility values specified by the customer for each requirement (in percentage)

	mpix	display	zoom	waterproof	movie	price
utility	21.0	9.0	30.0	7.0	8.0	25.0

stopped after a certain number of minimal conflict sets, it calculates all minimal conflict sets. This results in a tree where all paths to the leafs are either marked with *ok* or *closed*. All paths from the **root** node to a leaf marked with *ok* are minimal conflict sets. All other leafs are marked with *closed* are non-minimal conflict sets.

The performance improvements of the algorithm *BFX* compared to the *FastXplain* (see Section 3.3) can be explained by the way minimal diagnoses are selected.

3.5. Algorithm: Personalized FastXplain (PFX)

This section explains the *PFX (Personalized FastXplain)* algorithm which is an adaptation of the *FastXplain* algorithm (see Section 3.3) with the aim to personalize the calculation of minimal conflict sets. The *PFX* algorithm uses a similar data structure as *FastXplain* (see Table 3.2). The main difference between the two algorithms is that the *PFX* takes a utility value into account. The usage of the utility affects the ordering in which minimal conflict sets are retrieved.

In interactive settings there is no need to calculate all minimal conflict sets. In most settings it is enough to come up with a couple of alternatives. In this context a personalized ranking of the alternatives is important for the usability of the system. The *PFX* algorithm follows the idea that customers will more probably change requirements of low importance for them compared to requirements with a high utility (Mobasher, 2007). This results in the need for personalized conflict sets.

Identification of Personalized Conflict Sets

The *PFX* algorithm personalizes the calculation of conflict sets by using utility values i.e. values that indicate the importance for the customer. These utility values can be either entered by the customer or retrieved from the user profile. Table 3.4 shows the utility values (in percentage) for the example introduced in Section 3.1. Based on these utility values of a customer, the products can be ranked. The utility weight (see Definition 3.2) of a product p_i is the sum of all utility values from those attributes which are not satisfied by the product. In this formula v_j indicates the satisfaction value of the product p_i (also shown in Table 3.2) and the attribute a_j in the data structure. For example, let us take a look at product p_1. The product p_1 does not satisfy the requirements r_2 and r_6 (see Table 3.2). The satisfaction value v of these two requirements is 0 whereas the value of the other requirements is 1. Using these values and the utility values of Table 3.4 the utility weight of product p_1 can be calculated by: *(1-1)∗21 + (1-0)∗9 + (1-1)∗30 +*

(1-1)∗7 + (1-1)∗8 + (1-0)∗25 = 34. The utility weights of all products are shown in Table 3.5.

$$utilityweight(p_i) = \sum (1 - v_j) * utility(a_j) \tag{3.2}$$

Based on this, the utility for each product can be calculated. For all products of the working example the utility weights are shown in Table 3.5. This table shows that the product with the lowest utility weight is p_{11} and the derived diagnosis is $d_1 = \{r_5, r_6\}$. Similar to the *FastXplain* algorithm a directed acyclic graph (based on (Reiter, 1987)) can be built up using d_1. This graph is shown in Figure 3.5. For each element of the diagnosis, the algorithm can draw an edge starting from the root node. Following the edge $\{r_5\}$, the algorithm has to reconsider the set of requirements R (exclude $\{r_5\}$) and the set of products P (only products that do not satisfy r_5 are needed, i.e. $P_{[\neg r_5]}$). In addition, the weight for this node is calculated. This weight is the sum of all requirements of the path to this node (r_5 in this case). The weight of a requirement is based on the utility value (see Table 3.4). For example, the requirement r_5 concerns the attribute *movie*. This attribute has a utility weight of *8* (see Table 3.4). For this reason the utility of the requirement r_5 is $w = 8$. If the path consists of more requirements, the utility values of all requirements are summed up.

Table 3.5.: Utility values for each product (based on the Formula 3.2)

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}
utility-weight	34	61	84	36	41	34	36	66	45	46	33	83	34	54	64

After adding the diagnosis to the root node, there are two possibilities to expand the directed acyclic graph. The first one is to expand the node which has currently the lowest weight. This is highly dependent on the utility for the customer, because this type of search will not retrieve nodes with a minimal cardinality, but with a minimal utility weight. The second possibility is to perform a breath-first search where all nodes of the same level are expanded before going into a deeper level. Within one level the node with the lowest weight can be expanded first in order to improve the ranking of the resulting minimal conflict sets. In this thesis the second method is used because this enables us to ensure that all conflict sets that are calculated are minimal ones due to the characteristics of the breadth first search. This allows us to compare the approach with other algorithms (state of the art approaches as well as approaches introduced in this chapter).

For building up the directed acyclic graph, the algorithm uses the breadth-first search with the extension to expand the nodes with a lower weight first. Therefore the algorithm needs to calculate the weights for each node. For the node resulting from r_5 the weight is $w = 8$ and for the one resulting from r_6 the weight is $w = 25$. The algorithm starts expanding the node from r_5 first (see left part of the tree in Figure 3.5). The next diagnosis with the lowest utility weight is $d_2 = \{r_2, r_6\}$. After adding this diagnosis to the graph and adapting the set of requirements and the set of products it can be observed that the set of products is empty. This means that a minimal conflict set has been found, which is $\{r_5, r_6\}$. For the path $\{r_5 - r_2\}$ one product (p_{10}) is still left. Due to the breadth-first search, the algorithm continues with the node with the path $\{r_6\}$ (see right branch of Figure 3.5). The diagnosis with the lowest weight resulting from the product set $P_{[\neg r_6]} = \{p_4, p_7, p_8, p_9\}$ is $d_3 = \{r_1, r_4, r_5\}$. This diagnosis is added to the graph and when calculating

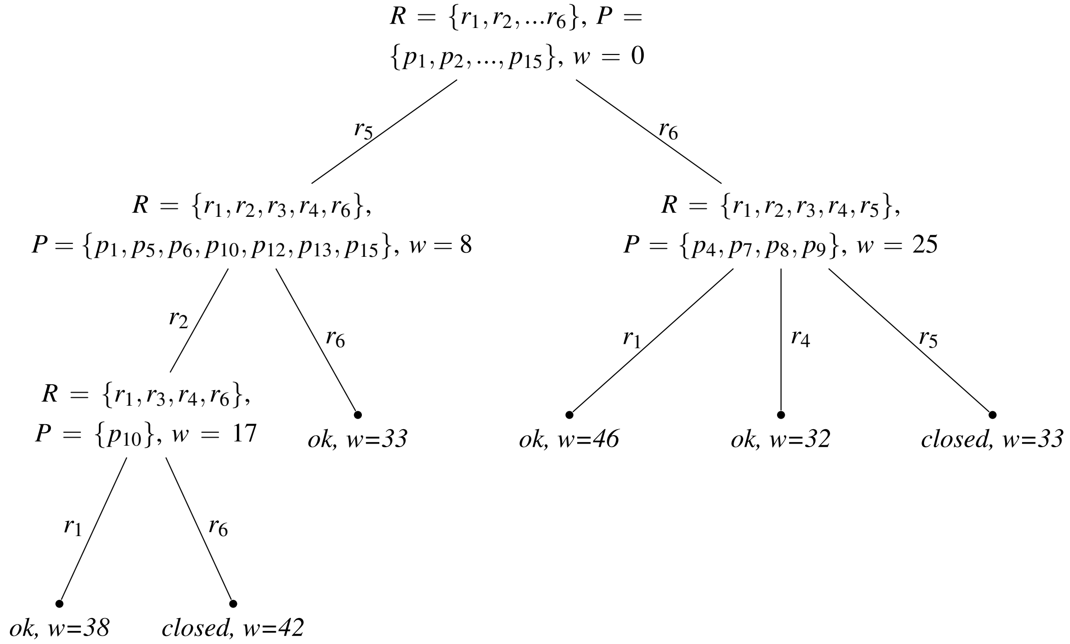


Figure 3.5.: Directed acyclic graph built of diagnoses to identify personalized conflict sets. The graph is constructed by the algorithm *PFX* (Personalized FastXplain) from the example of Section 3.1

the remaining set of products it can be observed that the algorithm identified two more minimal conflict sets ($\{r_6, r_1\}$ and $\{r_6, r_4\}$). The set $\{r_6, r_5\}$ would also be a minimal conflict set, but the algorithm has already found it. Therefore, the status of the path can be set to *closed*. Afterwards, the algorithm expands the last node with the diagnosis $d_4 = \{r_1, r_6\}$ which leads to the minimal conflict set $\{r_5, r_2, r_1\}$. Finally, all minimal conflict sets have been identified in a personalized ordering based on the utility values of Table 3.5: $MCS = \{\{r_5, r_6\}, \{r_1, r_6\}, \{r_4, r_6\}, \{r_1, r_2, r_5\}\}$.

Description of the *PFX* Algorithm

After describing the ideas behind the *PFX* (Personalized FastXplain) algorithm (see Algorithm 4), a more formal description is given now. The algorithm was inspired by *FastXplain* (see Section 3.3 and Schubert et al. (2010)). This section describes the main differences between the *FastXplain* algorithm and the *PFX* algorithm. The input values of the *PFX* algorithm are the **root** node for the graph, a set of customer requirements **R** and a set of products **P**. Compared to the *FastXplain* algorithm there is a fourth input value namely the utility value of each requirement. These values can be either specified by asking the user directly or retrieved from the user profile. Similar to the *FastXplain* algorithm the algorithm has a global variable *MCS* which holds all minimal conflict sets, but in a sorted (personalized) way. This variable is empty in the beginning.

The first step of the *PFX* algorithm is to retrieve the diagnosis with the lowest utility weight (for the calculation of the weight see Formula 3.2). Similar to the *FastXplain* algorithm, the *PFX* iterates over all requirements of the diagnosis and adds it to the graph. For each new node the set of requirements and

Algorithm 4 PFX(root, R, P, u)

```
{Input: root - the root node of the current subtree}
{Input: R - set of user requirements}
{Input: P - set of products}
{Global: MCS - set of all minimal conflict sets}
D ← getMinWeightDiagnosis(R, P)
for all requirements r from D do
   P' ← reduce(P, r)
   child ← addChild(r, R \ {r}, P')
   if P' = {} then
      if path(child) ∉ MCS then
         MCS ← path(child)
         child ← ok
         return
      end if
      child ← closed
      return
   end if
   if ∃ cs ∈ MCS : cs ⊆ path(child) then
      child ← closed
   else
      children ← sortInsert(child, u)
   end if
   PFX(children.nextChild(), R \ {r}, P')
end for
```

the set of products are adapted (similar to the *FastXplain* algorithm). Furthermore, it is checked if the path is a minimal conflict set (no products are left in the product set) or if the path is already a superset of any minimal conflict set found so far. If this is not the case the algorithm performs a sorted insert (**sortInsert(child, u)**) in the list of **children** in order to expand them in a personalized way. This list of children is always sorted according to the path length (length from the root node to the actual node) and the utility value **u**. The recursive call of the *PFX* algorithm is performed with the next child in the list (see $PFX(children.nextChild(), R \setminus \{r\}, P')$).

The main improvement of the algorithm *PFX* compared to the *FastXplain* (see Section 3.3) is that the minimal conflict sets are determined in a personalized way. In interactive settings it is often sufficient to propose only a few alternatives to the customer, but then these alternatives should be personalized in order to support the customer as good as possible.

3.6. Algorithm: PersRepair

This section explains the *PersRepair* algorithm which was published in (Felfernig et al., 2009b,c). Compared to the algorithms so far (*GraphXplain*, *FastXplain*, *BFX* and *PFX*) which are meant to calculate

minimal conflict sets, the goal of this algorithm is to calculate *repair actions*. A repair action is an action that is proposed to the customer in order to restore consistency of the given requirements. If the customer accepts a repair action, there is at least one item in the product assortment that satisfies the adapted requirements of the customer. This adaptation is called *repair action*. Due to the fact that repair actions ignore the original customer requirements at least partially, there is no information if a repair alternative is of relevance to the customer. Furthermore, as there exists a repair action for each item, the set of repair actions gets potentially very large. Consequently, there exists a need for a personalization of repair alternatives.

Compared to the algorithms (*GraphXplain*, *FastXplain*, *BFX* and *PFX*) described in this chapter so far, the *PersRepair* algorithm does not take the structural properties of a knowledge-based recommender into account (the algorithm can also be applied to general Constraint Satisfaction Problems, CSP). The *PersRepair* algorithm applies concepts of model-based diagnosis (Reiter, 1987; de Kleer et al., 1992) for the automated identification of minimal sets of faulty requirements. If no recommendation can be identified (as it is the case in the example given), the diagnosis task is to determine those requirements that need to be relaxed or deleted in order to find a recommendation. The possible adaptations or relaxations of the customer requirements are denoted as repair actions. These repair actions are based on diagnoses which have to be calculated beforehand.

Identifying Repair Actions

The goal of the *PersRepair* algorithm is to systematically reduce the number of repair actions. The reduction of the repair actions is based on the idea of taking into account only the set of nearest neighbors (already completed recommendations that are similar to the requirements of the current user). A special interest lies on those repair actions which resemble the original requirements of the customer the most. For deriving such repair actions, the existing product definitions (see Table 3.1) are exploited and the n-nearest neighbours are identified. The determination of the similarity values is based on attribute-level similarity measures (Konstan et al., 1997; Wilson and Martinez, 1997; McSherry, 2004). The similarity is calculated for each pair of attribute a_i of the product assortment P and the corresponding customer requirement r_i. Depending on the characteristics of the attribute, one of the following three measures is taken: *More-Is-Better* (MIB) see Formula 3.3, *Less-Is-Better* (LIB) see Formula 3.4 or *Nearer-Is-Better* (NIB) see Formula 3.5 (McSherry, 2004). In these formulas $val(r_i)$ represents the value of the requirement r_i; $max(a_i)$ is the maximum value of the attribute a_i in the product assortment and $min(a_i)$ is the minimal value.

$$MIB : sim(r_i, a_i) = \frac{val(r_i) - min(a_i)}{max(a_i) - min(a_i)} \tag{3.3}$$

$$LIB : sim(r_i, a_i) = \frac{max(a_i) - val(r_i)}{max(a_i) - min(a_i)} \tag{3.4}$$

$$NIB : sim(r_i, a_i) = \begin{cases} 1 & if\ val(r_i) = val(a_i) \\ 0 & else \end{cases} \tag{3.5}$$

For the working example the MIB similarity is used for the attributes *mpix*, *display*, *opt-zoom* and *weight*. The NIB similarity is used for the attributes *sound*, *waterproof*, *movies* and *colour*. For the attribute *price*

the LIB similarity is used. Calculating the similarity between the attribute *mpix* of the product p_1 and the customer requirement r_1 ($mpix > 10$), the maximum $max(mpix) = 15$ and the minimum $min(mpix) = 5.5$ are taken as a basis. Based on the characteristics of the attribute *mpix* the MIB similarity is applied. Therefore, the similarity of this attribute is $sim(r_1, mpix) = \frac{10-5.5}{15-5.5} = 0.47$. The idea behind the MIB similarity is the higher the value the better it is for the customer. In comparison to this the attribute *price* holds the following: the lower the value the better it is for the customer (LIB). For a detailed discussion of different types of similarity measures see, for example, (Wilson and Martinez, 1997; McSherry, 2004).

Based on these individual similarity values for each attribute, the similarity between the customer requirements and the product under consideration can be defined. Formula 3.6 calculates the overall similarity value. In this formula, $weight(r_i)$ denotes the *importance* of requirement r_i for the current user. For the working example, it is assumed that all requirements are equally important to the customer. The weight for each requirement r_i is $weight(r_i) = \frac{1}{6}$ ($w(mpix) = \frac{1}{6}$, $w(display) = \frac{1}{6}$, $w(opt-zoom) = \frac{1}{6}$, $w(waterproof) = \frac{1}{6}$, $w(movies) = \frac{1}{6}$ and $w(price) = \frac{1}{6}$). For the working example introduced in Section 3.1 the similarities between the customer requirements and all products are shown Table 3.6. Based on these similarity values, the n-nearest neighbours can be determined. For the working example $n = 5$ has been chosen, which results in the following 5-nearest neighbours (the ones with the highest similarity): $NN = \{p_2, p_6, p_{10}, p_{13}, p_{15}\}$.

$$similarity(R, p_j) = \sum_{r_i \in R} sim(r_i, a_i) * weight(r_i) \tag{3.6}$$

The *PersRepair* algorithm (see Algorithm 5 and 6) identifies personalized repair actions based on a set of nearest neighbours **NE**. The initial situation is similar to algorithms already described. The set of requirements R is inconsistent with the set of products. Therefore the customer requirements are also inconsistent with the set of nearest neighbours $NN = \{p_6, p_7, p_{10}, p_{13}, p_{15}\}$ ($\sigma_{[R]}NN = \emptyset$). In order to help the customer in consistency management, the *PersRepair* algorithm activates a conflict detection component which returns one conflict set per activation. For the implementation of this algorithm, the algorithm *QuickXplain* (Junker, 2004) was used. Other algorithms can be used as well, for example, the ones described earlier in this chapter. For all products in *NN* and the requirements in R the following conflict sets can be derived: $CS = \{\{r_1, r_6\}, \{r_4, r_6\}, \{r_5, r_6\}, \{r_1, r_2, r_5\}\}$.

Table 3.6.: Similarity values for each product (based on the Formula 3.6)

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}
similarity	.46	**.59**	.46	.34	.41	**.59**	.53	.53	.44	**.63**	.49	.11	**.62**	.44	**.62**

The conflict set $CS_1 = \{r_1, r_6\}$ is the first one retrieved. Using this conflict set, the construction of the Hitting Set Directed Acyclic Graph (HSDAG) (Reiter, 1987) can be started (see Figure 3.6 for the whole graph). The first conflict set that is retrieved is added to the root node. The conflict set is represented by two outgoing paths, namely $\{r_1\}$ and $\{r_6\}$. Following the edge of the graph indicates the relaxation of this requirement (label of the edge). Based on the situation after adding the first conflict set, it is analysed

which repair actions are possible after eliminating one element from this conflict set. If, for example, the requirement r_1 is eliminated this would result in repair actions supported by $NN_{[\neg r_1]} = \{p_2, p_{10}\}$. The algorithm continues with expanding the tree in a best first manner that follows the highest similarity value of the first i products (for this example $i = 3$ is used). The best first search leads to repair actions that are most similar to the original customer requirements. In the example introduced, the path $\{r_6\}$ is the most promising one. Therefore the second conflict set $CS_2 = \{r_1, r_2, r_5\}$ is added to this branch. After adding this conflict set to the graph three diagnoses can be found: $d_1 = \{r_6, r_2\}$, $d_2 = \{r_6, r_1\}$ and $d_3 = \{r_6, r_5\}$. The ordering of these diagnoses is determined on the basis of the similarity values determined for the set of nearest neighbours. For example, for the path $\{r_6 - r_2\}$ the system can suggest repair actions on the basis of the products $\sigma_{[\neg r_6, \neg r_2]NN} = \{p_2, p_6, p_{10}\}$.

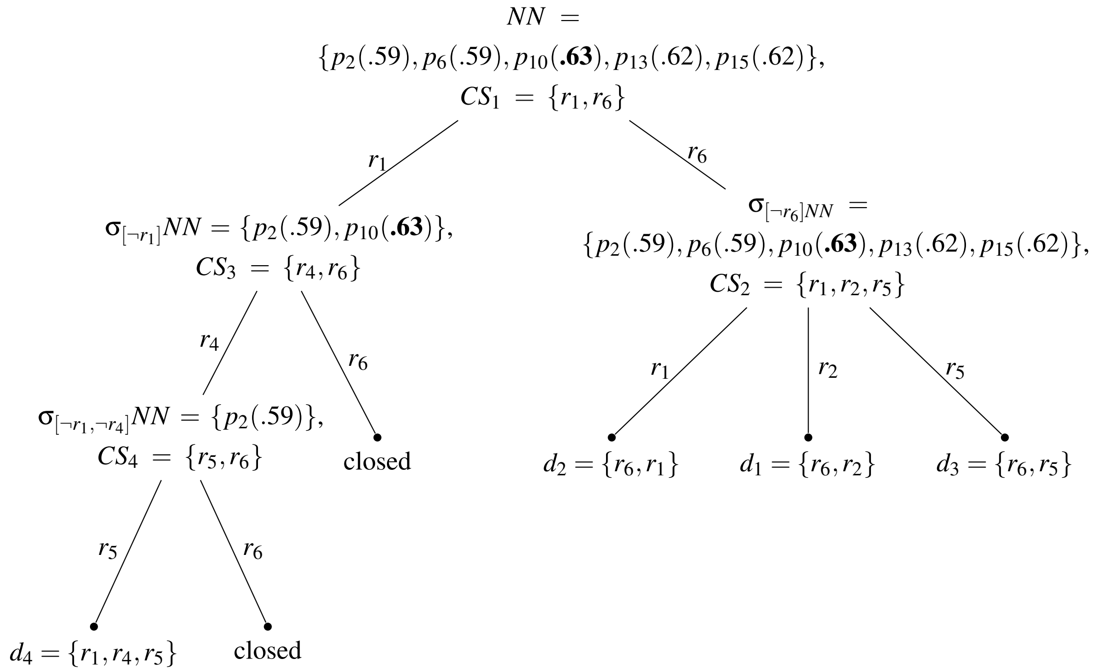


Figure 3.6.: Directed acyclic graph built of conflict sets to identify personalized diagnoses by the algorithm *PersRepair*. The algorithm operates on the 5 nearest neighbour products (*NN*). The value close to the product id indicates the similarity between the product and the customer requirements (see Table 3.6).

The key idea behind this approach is that the most promising paths are leading to those diagnoses that are most similar to the original set of requirements. In order to follow the most promising path the algorithm expands the graph in a best-first manner. The complete graph is shown in Figure 3.6. In order to calculate the repair actions the algorithm takes the diagnosis d_1. This is the one that is most similar to the original requirements. Based on the diagnoses the repair actions can be directly derived from the product assortment. For each attribute of a diagnosis the algorithm needs to identify a possible adaptation that leads to at least one product. The repair actions for diagnosis d_1 are based on $\sigma_{[R-d_1]}NN = \{p_6, p_{13}\}$. The calculation is the following:

$$\pi_{[attributes(d_1)]}(\sigma_{[R-d_1]}NN) = \pi_{[attributes(\{r_2,r_6\})]}(\sigma_{[\{r_1,r_3,r_4,r_5\}]}NN) = \pi_{[attributes(\{r_2,r_6\})]}(\{p_6, p_{13}\})$$

The set of possible repair actions for the diagnosis $d_1 = \{r_2, r_6\}$ are $\{\{display = 2.8, price = 249\}, \{display = 2.5, price = 179\}\}$.

Description of the *PersRepair* Algorithm

The algorithm *PersRepair* is shown in Algorithm 5. This algorithm highly depends on the *PersDiagnosis* algorithm (see Algorithm 6). After showing how both algorithms work using the working example, a more formal description follows. The *PersRepair* algorithm takes the following parameters as input: **R** which is the set of requirements specified by the customer, **P** the description of the product assortment (this can be a product table similar to Table 3.1), with **n** one can specify how many nearest neighbours should be used and with **k** one can specify how many similar items should be considered for the best-first search criteria. Based on the requirements **R** and the products **P**, the algorithm calculates **n** nearest neighbours by the function **getNearestNeighbours(R,P,n)**. The result is stored in the variable **NN** and the *PersDiagnosis* algorithm (see Algorithm 6) is called for calculating one personalized diagnosis. Based on the diagnosis of the *PersDiagnosis* algorithm, different repair actions may be retrieved from the set of nearest neighbours. These repair actions are returned and can be presented to the user.

Algorithm 5 PersRepair(R, P, n, c)

{Input: R - set of user requirements}
{Input: P - set of products}
{Input: n - number of nearest neighbours}
{Input: k - k most similar items to be used by *SimilaritySort(H,k)*}
{Output: repairs - sorted list of repair actions}
$NN \leftarrow getNearestNeighbours(R, P, n)$
$d \leftarrow PersDiagnosis(R, NN, \emptyset, k)$
return $\pi_{attributes(d)}(\sigma_{[R-d]} NN)$

The algorithm *PersDiagnosis* is shown in Algorithm 6. This algorithm takes the set of customer requirements **R** and the nearest neighbours **NN**. The parameter **H** is a bag structure which is initiated with $\emptyset$. This structure holds all paths of the search tree in a best-first fashion. The last parameter **k** specifies how many products should be taken into account by the best-first search. The algorithm starts to retrieve the current best path **d**. This is the one with the most promising repair alternatives, meaning the one with the highest probability to be adopted by the customer. The goal is to find those repair alternatives which are most similar to the original requirements in **R**. Based on the current requirements (all requirements except the path **R-d**) and the nearest neighbours the algorithm calls a conflict detection algorithm (**ConflictDetectionAlgorithm(R-d,NN)**). The role of this algorithm is to check whether there exists a product for the current set of reduced requirements (**R-d**). If there exists at least one product that satisfies the reduced set of requirements, then the call of the conflict detection approach returns an empty set. This means that the algorithm has found a diagnosis (**d**). If the conflict detection algorithm identifies a conflict set (**ConflictDetectionAlgorithm(R-d,NN)** returns a non-empty set **CS**), then the algorithm needs to expand the node resulting from the current path **d**. For each element **r** from the conflict set the algorithm adds a new path $\{d \cup \{r\}\}$ to the bag structure **H**. Finally, after adding the new elements to **H**, the algorithm has to make sure that it is sorted according to the sorting criteria **k** (**SimilaritySort(H,k)**). The algorithm

continues to expand the paths further until one diagnosis has been found. It can also be adapted to calculate *m* or *all* diagnoses. If all diagnoses for the working example have been calculated, the graph would look like the one in Figure 3.6.

Algorithm 6 PersDiagnosis(R, NN, H, k)

$d \leftarrow first(H)$
$CS \leftarrow ConflictDetectionAlgorithm(R - d, NN)$
if $CS = \emptyset$ **then**
  return d
end if
for all *requirements r from CS* **do**
  $H \leftarrow H \cup \{d \cup \{r\}\}$
end for
$H \leftarrow SimilaritySort(H, k)$
$PersDiagnosis(R, NN, H, k)$

3.7. Algorithm: ReAction

This section introduces the algorithm *ReAction* which was published in (Schubert et al., 2011). The task of the algorithm is to support customers with preferred repair actions efficiently. This results in two main goals: First, the repair actions should be personalized and second, these repair actions should be calculated fast, because a suitable run time performance is important for interactive settings (see Section 1.2).

Identifying Preferred Repair Actions

First, the term *preferred repair actions* will be clarified. Preferred repair actions result from preferred diagnoses. In order to define a preferred diagnosis, this work relies on the definition of a total ordering of the given set of requirements in *R*. This ordering of the requirements can be retrieved by asking customers directly regarding their preferences. Another option is to apply multi attribute utility theory (MAUT) (von Winterfeldt and Edwards, 1986). Moreover, (Belanger, 2005) introduced a conjoint analysis which can be used to determine a ranking between the requirements. The preference values specified by the customer in the working example introduced in Section 3.1, are shown in Table 3.4.

For this work the following definition of a lexicographical ordering (Definition 8) is used. This definition is based on the total ordering for constraints that has been applied in (Junker, 2004) for the determination of preferred conflict sets.

Definition 8 *Total Lexicographical Ordering: Given a total order $<$ on R, we enumerate the requirements in R in increasing $<$ order $r_1, ..., r_n$ starting with the least important requirement (i.e., $r_i < r_j \Rightarrow i < j$). We compare two subsets X and Y of R lexicographically:*

$$X >_{lex} Y \text{ iff}$$

$$\exists k : r_k \in Y - X \text{ and}$$
$$X \cap r_{k+1}, ..., r_t = Y \; r_{k+1}, ..., r_t.$$

This definition of the total lexicographical ordering is based on the work of (Junker, 2004). (Junker, 2004) uses this definition for the determination of preferred conflict sets. This definition can be adapted in order to define a preferred diagnosis (see Definition 9):

Definition 9 *A minimal diagnosis d is a preferred diagnosis iff there does not exist another minimal diagnosis d' with $d' >_{lex} d$.*

The first step of the algorithm *ReAction* is to rank the requirements of the customer in a lexicographical order. For the working example (see Section 3.1), the utility values from Table 3.4 are applied which results in the lexicographical ordering ($r_4 < r_5 < r_2 < r_1 < r_6 < r_3$). The least important requirement of the customer is r_4 (attribute *stabilization*) and the most important requirement is r_3 (attribute *zoom*). If all customer requirements are considered, then a conflict situation emerges (no product can be found for the requirements in *R*). The algorithm *ReDiagnosis* is called with the sorted set of requirements. This algorithm returns the preferred diagnosis, meaning the one including the attributes with the lowest overall utility. The underlying assumption is that this diagnosis is the one that will most probably to be changed by the user (Junker, 2004).

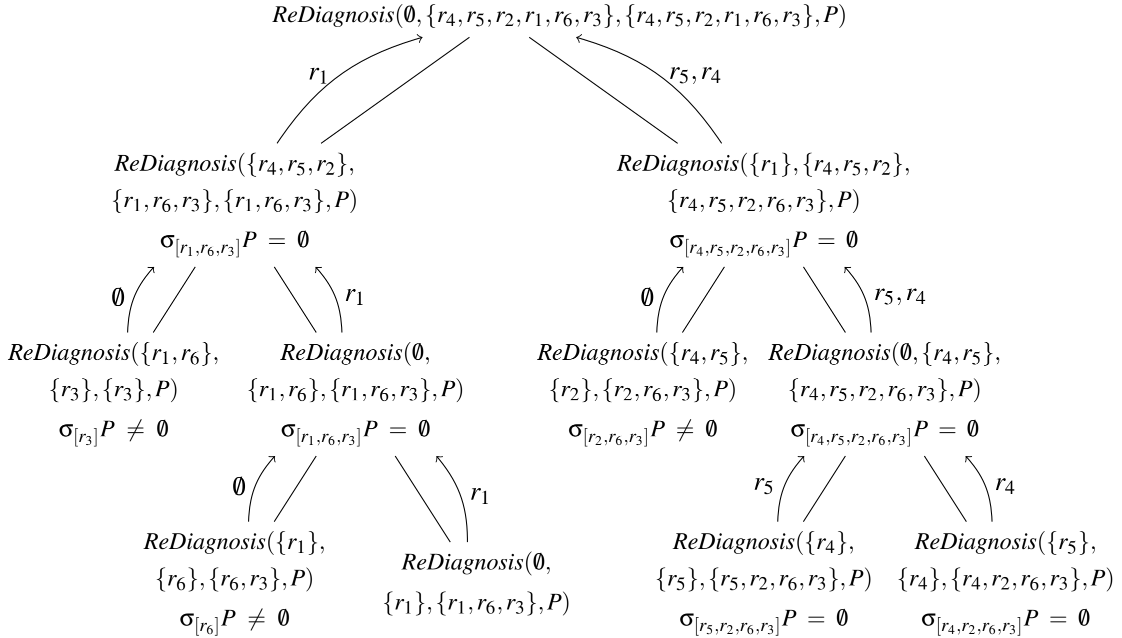


Figure 3.7.: Execution tree of the *ReDiagnosis* algorithm to determine one preferred diagnosis. The algorithm is called with a sorted set of requirements $R = \{r_4, r_5, r_2, r_1, r_6, r_3\}$ and the product table P

The *ReDiagnosis* algorithm was inspired by the *FastDiag* algorithm which is described in Section 4.2 in the context of configuration systems. The *ReDiagnosis* algorithm is called with a delta **D** which is

empty in the beginning ($\mathbf{D} = \emptyset$). This delta is needed to store the requirements that split from the current set of requirements. The next input parameter of the algorithm is a set of requirements that should be diagnosed $\mathbf{R} = \{r_4, r_5, r_2, r_1, r_6, r_3\}$. Finally a set of all requirements ($\mathbf{AR} = \{r_4, r_5, r_2, r_1, r_6, r_3\}$) which can be larger compared to **R** is needed as well as the product assortment **P**. The whole execution tree is shown in Figure 3.7. As the algorithm continues, the original set of requirements $R = \{r_4, r_5, r_2, r_1, r_6, r_3\}$ is split into two parts $R_1 = \{r_4, r_5, r_2\}$ and $R_2 = \{r_1, r_6, r_3\}$. In a recursive way the algorithm exploits the customer requirements. In the next step the algorithm is again called with the subsets R_1 and R_2 (*ReDiagnosis*($\{r_4, r_5, r_2\}$,$\{r_1, r_6, r_3\}$,$\{r_1, r_6, r_3\}$,*P*)). Then it is checked whether the query $\sigma_{[r_1,r_6,r_3]}P$ returns any product. This is not the case as $\sigma_{[r_1,r_6,r_3]}P = \emptyset$. Therefore, the algorithm has to split up the current set ($R'_1 = \{r_6, r_1\}$ and $R'_2 = \{r_3\}$) and calls itself again. $\sigma_{[r_3]}P = \{p_1, p_2, p_4, ...\} \neq \emptyset$ returns more than one product. This retrieved set of products ($\{p_1, p_2, p_4, ...\}$) indicates, that there is not any conflict in this set of requirements (see also Definition 4). For this reason, the algorithm does not need to further investigate in this direction, i.e. an empty set can be returned. The next step is to look into the set $R = \{r_1, r_6\}$ and $AR = \{r_1, r_6, r_3\}$. The query with AR ($\sigma_{[r_1,r_6,r_3]}P = \emptyset$) does not return any product so the algorithm need to split the query up again into $\{r_1\}$ and $\{r_6, r_3\}$. The call with $R = \{r_1\}$ returns r_1 which means that at least r_1 is an element of the diagnosis. This element of the diagnosis is returned and we can follow the right branch of the execution tree (see Figure 3.7). This execution trace follows the same principle and finally returns the set $\{r_5, r_4\}$. This means that a diagnosis $d_1 = \{r_1, r_5, r_4\}$ has been calculated, which is the one that is most likely to be changed by the customer.

Description of the *ReAction* Algorithm

This section gives a more formal description of the algorithms *ReAction* and *ReDiagnosis*. The algorithm *ReAction* is shown in Algorithm 7. This algorithm uses the algorithm *ReDiagnosis* (see Algorithm 8) to calculate minimal diagnoses. The algorithm *ReAction* takes the customer requirements $\mathbf{R} = \{r_1, r_2, ..., r_n\}$, the product assortment $\mathbf{P} = \{p_1, p_2, ..., p_m\}$ as well as a sorting criteria **c** as inputs.

Algorithm 7 ReAction(R, P, c)

{Input: R - customer request (requirements)}
{Input: P - table of all products}
{Input: c - sort criteria (i.e. utility)}
{Output: repairs - sorted list of repair actions}
if $\sigma_{[R]}P \neq \emptyset$ **then**
  return $\emptyset$
end if
$R \leftarrow sort(R, c)$
$d \leftarrow ReDiagnosis(\emptyset, R, R, P)$
$repairs \leftarrow \Pi_{[attributes(d)]}(\sigma_{[R-d]}P)$
return *repairs*

The first step of the algorithm is to check whether there exists a conflict situation. A conflict situation appears if no product from the assortment satisfies all customer requirements. This check is done by a query to the product assortment that includes all requirements in R ($\sigma_{[R]}P \neq \emptyset$). If there exists at least

one product of the assortment, the algorithm *ReAction* returns an empty set, because no repair action can be calculated. The next step the algorithm takes, is to sort the requirements of the customer according to the sorting criteria (**sort(R,c)**). As sorting criteria **c**, different personalization strategies can be used (i.e. similarity, utility, probability, and hybrids). This sorting criteria incorporates also the weights comming from the customer, from the multi attribute theory, or the conjoint analysis. This work focuses on the utility values that are assigned by the customer. Sample utility values for the example are shown in Table 3.5. If there exists a conflicting situation (no product can be found in the product assortment P for the customer request R), the algorithm *ReDiagnosis* (see Algorithm 8) is called with the ordered set of requirements. The *ReDiagnosis* algorithm returns one preferred minimal diagnosis. This is the diagnosis including the attributes with the lowest overall utility as these are the most probable ones to be changed by the customer (Junker, 2004). Based on such a diagnosis are calculated and returned.

Algorithm 8 ReDiagnosis(D, R, AR, P)

{Input: D - delta set, initially empty}
{Input: R - customer request (requirements) $\{r_1, r_2, ..., r_n\}$}
{Input: AR - all customer requirements (initial same as R)}
{Input: P - table of all products}
{Output: diagnosis - set of faulty requirements}
if $D \neq \emptyset$ **and** $\sigma_{[AR]}P \neq \emptyset$ **then**
  return $\emptyset$
end if
if $singleton(R)$ **then**
  return R
end if
$k \leftarrow \lceil \frac{n}{2} \rceil$
$R_1 \leftarrow \{r_1, ..., r_k\}$
$R_2 \leftarrow \{r_{k+1}, ..., r_n\}$
$\delta_1 \leftarrow ReDiagnosis(R_1, R_2, AR - R_1, P)$
$\delta_2 \leftarrow ReDiagnosis(\delta_1, R_1, AR - \delta_1, P)$
return $(\delta_1 \cup \delta_2)$

The algorithm *ReDiagnosis* follows a *divide and conquer* strategy. It takes a delta **D**, which is empty in the beginning, a sorted set of requirements **R** as well as a set of all requirements **AR**. This set (**AR**) can be larger than the set of requirements (**R**), if there are some additional constraints that should be considered (coming, for example, from a knowledge base). The last parameter of the algorithm is the product assortment **P**. When the delta **D** is not empty and a query with all elements of the current set of all requirements **AR** returns at least one product, then we know that there cannot be a diagnosis in this set and, for this reason, the algorithm returns an empty set $\emptyset$. Moreover, the set of requirements **R** need to be split up for finding the minimal diagnosis. If the set of requirements **R** contains only one single element (**singleton(R)**), it cannot be split up any more. If only one element is left (after the consistency check), then this requirement is part of the diagnosis and must be returned (**return R**). An example for this is the call of *ReDiagnosis(*$\emptyset$*,* $\{r_1\}$*,* $\{r_1, r_6, r_3\}$*, P)*. In this call the current set of requirements (**R** = $\{r_1\}$) is part of the diagnosis, because the query with **AC** ($\sigma_{[r_1, r_6, r_3]}P$) returns an empty set. If the set of requirements **R** contains more than one element, then it is split in the middle ($\mathbf{R}_1 \leftarrow \{r_1, ..., r_k\}$ and

$\mathbf{R}_2 \leftarrow \{r_{k+1},...,r_n\}$). With these two adapted sets the *ReDiagnosis* algorithm can call itself in a recursive way. Both recursive calls of the algorithm can lead to parts of the diagnosis. Moreover, the algorithm has to return the aggregation of both (**return** $(\delta_1 \cup \delta_2)$). Based on this diagnosis, the algorithm *ReAction* can retrieve possible repair actions that are valuable to the customer.

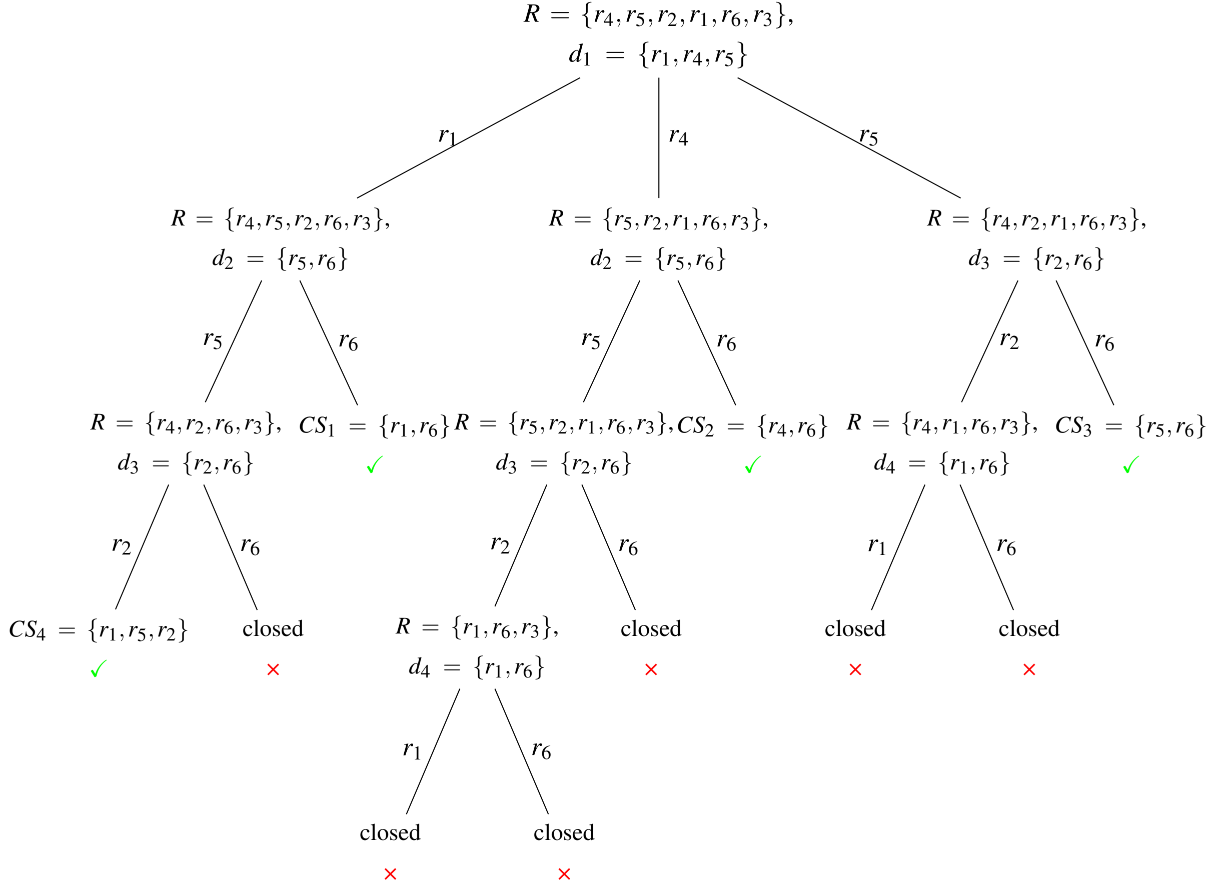


Figure 3.8.: Directed acyclic graph constructed by the algorithm *ReDiagnosis* for calculating more than one diagnosis. Similar to the *FastXplain* algorithm, the minimal conflict sets can be retrieved from the graph ($CS_1 = \{r_1, r_6\}$, $CS_2 = \{r_4, r_6\}$, $CS_3 = \{r_5, r_6\}$, $CS_4 = \{r_1, r_5, r_2\}$)

Calculating More Repair Sets

The *ReDiagnosis* algorithm (Algorithm 8) can be adapted in order to calculate more than one diagnosis. Inspired by the hitting set directed acyclic graph (HSDAG) (Reiter, 1987), the adapted algorithm *ReDiagnoses-Tree* builds up a tree based on diagnoses. Basically, in each step of the algorithm one diagnosis can be identified using *ReDiagnosis*. For this diagnosis, the edges may be added to the current node. Figure 3.8 shows the tree for the working example. The first diagnosis retrieved from *ReDiagnosis* is $d_1(R) = \{r_1, r_4, r_5\}$. This diagnosis is added to the root node and for each element of the diagnosis one edge is added to the root node. In the next step, the next diagnosis is calculated for the left path -

namely for the requirements except the path $(R - r_1)$. The call of the *ReDiagnosis* algorithm with $D = \emptyset$, $R = \{r_4, r_5, r_2, r_6, r_3\}$, $AR = \{r_4, r_5, r_2, r_1, r_6, r_3\}$ and P returns the next diagnosis $d_2 = \{r_5, r_6\}$. This strategy is continued in a breadth-first manner. If the algorithm cannot find a diagnosis for a leaf any more ($R - path$ is consistent), this leaf is closed. The whole algorithm continues until all leafs are closed or a given number of diagnoses is reached.

3.8. Evaluation

This section presents the outcomes of the evaluation. In this evaluation the techniques and algorithms introduced in this chapter are compared with other state-of-the-art approaches. The evaluation section is divided into two parts, namely the performance evaluation and the quality evaluation. In the *performance evaluation*, the run time performance of the different approaches is evaluated. The run time is mainly influenced by the number of customer requirements and number of products in the assortment. Additionally, the *satisfaction rate* (relation between the number of fulfilled requirements to the number of requirements, see also Formula 3.7) has an impact on the run time. In the *quality evaluation*, the evaluation results regarding the acceptance probability (precision) for the customer, are presented.

The aim of the evaluation is to relate the performance of the different introduced approaches to each other as well as to state-of-the-art approaches. Besides the algorithms introduced, three state-of-the-art algorithms have been implemented. The first one is a combination of the *QuickXplain* (Junker, 2004) and the hitting set directed acyclic graph (*HSDAG*) (Reiter, 1987). This combination was also described in Section 1.1 and 3.6 and from now on it will be referred as *QuickXplain*. This algorithm performs a breadth-first search in the *HSDAG* and uses the *QuickXplain* as a theorem prover. Note that also the pruning functionality (Greiner et al., 1989) has been implemented into our *HSDAG*. The second approach used as comparison for the introduced algorithms is the *CorrectiveRelax* algorithm by (O'Callaghan et al., 2005). This algorithm aims to calculate *corrective explanations*. A corrective explanation is the complement of a diagnosis (relaxation). The algorithm *CorrectiveRelax* (O'Callaghan et al., 2005) was inspired by the algorithm *QuickXplain* (Junker, 2004). Compared to the *QuickXplain* which calculates minimal conflict sets using the divide-and-conquer principle, the *CorrectiveRelax* algorithm calculates corrective explanations (similar to diagnoses) based on a binary search. These relaxations are found by removing, one-by-one, constraints that cause inconsistency. Initially the maximal relaxation is empty. In each step the *CorrectiveRelax* algorithm adds constraints for which the consistency is ensured (i.e. at least one solution exists). The constraints that are responsible for the inconsistency are continuously removed. In the following the execution of the *CorrectiveRelax* is shown for the example given in Section 3.1:

- **CorrectiveRelax**($\emptyset$, $\{r_1, r_2, r_3, r_4, r_5, r_6\}$, $\emptyset$, P)
  note that the product table P is used for the consistency check (evaluating if a product can be retrieved from the assortment)
  as the empty set ($\emptyset$) is *consistent* with P, the requirements are split up to $\{r_1, r_2, r_3\}$ and $\{r_4, r_5, r_6\}$
- **CorrectiveRelax**($\{r_1, r_2, r_3\}$, $\{r_4, r_5, r_6\}$, $\{r_1, r_2, r_3\}$, P)
  as the set $\{r_1, r_2, r_3\}$ is *consistent* with P, the requirements ($\{r_4, r_5, r_6\}$) are split up to $\{r_4, r_5\}$ and $\{r_6\}$

- **CorrectiveRelax**($\{r_1, r_2, r_3, r_4, r_5\}$, $\{r_6\}$, $\{r_4, r_5\}$, P)
  as the set $\{r_1, r_2, r_3, r_4, r_5\}$ is *not consistent* with P, the set storing the last changes ($\{r_4, r_5\}$) needs to be split up to $\{r_4\}$ and $\{r_5\}$
- **CorrectiveRelax**($\{r_1, r_2, r_3, r_4\}$, $\{r_5, r_6\}$, $\{r_4\}$, P)
  as the set $\{r_1, r_2, r_3, r_4\}$ is *consistent* with P, the requirements ($\{r_5, r_6\}$) are split up to $\{r_5\}$ and $\{r_6\}$
- **CorrectiveRelax**($\{r_1, r_2, r_3, r_4, r_5\}$, $\{r_6\}$, $\{r_5\}$, P)
  as the set $\{r_1, r_2, r_3, r_4, r_5\}$ is *not consistent* with P, and the set of current requirements contains only one element, the algorithm splits up the set requirements that yet to be tested ($\{r_6\}$) to $\{r_6\}$ and $\emptyset$
- **CorrectiveRelax**($\{r_1, r_2, r_3, r_4, r_6\}$, $\emptyset$, $\{r_6\}$, P)
  as the set to be tested is empty, the algorithm returns $\{r_1, r_2, r_3, r_4\}$ as a maximal relaxation

As shown the *CorrectiveRelax* algorithm identifies one minimal relaxation $\{r_1, r_2, r_3, r_4\}$. This relaxation the complement of the corresponding diagnosis $\{r_5, r_6\}$. Another important issue is that the prediction accuracy of *CorrectiveRelax* depends on the ordering of the requirements. Note that in this example, we did not apply any intelligent ordering to the requirements, except the index. The basic algorithm just identifies one corrective explanation. Nevertheless, it can be combined with the *HSDAG* approach by (Reiter, 1987) (similar to the *ReAction* algorithm, see also the description in (O'Callaghan et al., 2005)) . This allows us to calculate all corrective explanations and/or the corresponding diagnoses.

The third approach used for comparison, is the *MinRelax* introduced by (Jannach, 2008). This algorithm aims to find *maximal succeeding sub queries* (XSS). This is done by evaluating the individual sub queries of the original query independently in advance. The set of all XSS is generated by combining these partial results. This algorithm can only calculate all maximal succeeding sub queries at a time, which is not explicitly needed in interactive systems. Note that the complement of a maximal succeeding sub query is a minimal diagnosis. For a better understanding, *MinRelax* is applied to the first five products (p_1, p_2, p_3, p_4, and p_5) and all requirements of the example introduced in Section 3.1. For each product of the assortment *MinRelax* calculates a product specific relaxation, which is then checked if it is minimal.

- **Product $\mathbf{p_1}$**: the relaxation is $PSX_1 = \{r_2, r_6\}$;
  as it is the first one, there does not exist a super set of it
- **Product $\mathbf{p_2}$**: the relaxation is $PSX_2 = \{r_1, r_4, r_5, r_6\}$;
  this set is not a superset of PSX_1, nor is PSX_1 a superset of PSX_2 i.e. PSX_2 can be stored as a relaxation
- **Product $\mathbf{p_3}$**: the relaxation is $PSX_3 = \{r_1, r_3, r_5, r_6\}$;
  this set is not a superset of any relaxation derived so far, additionally is neither PSX_1 nor PSX_2 a superset of PSX_3 i.e. it can be stored as a relaxation
- **Product $\mathbf{p_4}$**: the relaxation is $PSX_4 = \{r_1, r_4, r_5\}$;
  this is a subset of PSX_2, for this reason PSX_2 needs to be deleted from the set of minimal relaxations
- **Product $\mathbf{p_5}$**: the relaxation is $PSX_5 = \{r_2, r_4, r_6\}$;
  this is a superset of PSX_1, i.e. PSX_5 is not minimal

The minimal relaxations derived from the first five products of the assortment are PSX =$\{r_2, r_6\}$, $\{r_1, r_3, r_5, r_6\}$, and $\{r_1, r_4, r_5\}$ as calculated above.

Some of the introduced approaches, aim to calculate minimal conflict sets and others aim to calculate minimal diagnoses. The dualities between conflicts, relaxations and diagnoses have already been pointed out by the example given in Section 1.1 as well as in Section 2.3. A further discussion regarding the interrelation of conflicts, diagnoses and relaxations can be found in (Junker, 2004). The fact that diagnoses can be derived from conflicts sets and vice versa, helps us make the approaches comparable. Nevertheless, in the performance evaluation, the focus lies on the identification of minimal conflict sets as well as on the calculation of minimal diagnoses. If an algorithm does not calculate minimal conflict sets such as the algorithms *ReAction*, *MinRelax* and *CorrectiveRelax*, the concepts of model based diagnoses (Reiter, 1987) are used to identify the minimal conflict sets based on diagnoses (for a detailed explanation see Section 3.3). The algorithm *MinRelax* is only dedicated to calculate all maximal succeeding sub queries. Therefore the *MinRelax* is not included in the evaluation where only few diagnoses or conflict sets are calculated.

In the following sections all approaches are evaluated according to their run time performance (see Section 3.8.1) and to their prediction quality (see Section 3.8.2).

3.8.1. Performance Evaluation

All algorithms described in this chapter can be applied in interactive recommender systems. As the customer interacts with the system, a crucial point is a good run time performance due to the time-bound. According to (Miller, 1968; Card et al., 1991) the limits for a system to react are the following: If a system responds in less then **0.1 second** the user feels that the system is reacting instantaneously. The limit for the user's flow of thought to stay uninterrupted is **1.0 second**. Although the user will notice the delay of the system, there is no need for a special feedback, if the response time of a system lies between **0.1** and **1.0 second**. If the system takes longer then **1.0 second**, the user loses the feeling on operating on the data. Therefore, it is really crucial that the algorithms can respond fast.

A performance analysis was conducted to show the applicability of the different approaches. First, the approaches are analysed using different settings. In this analysis it is discussed which algorithm is suited for which setting. Afterwards the results of the performance analysis with different test settings are presented. All algorithms[*] are implemented in Java 1.6 and all experiments have been executed on a desktop PC (*Intel®Core™2 Quad CPU Q9400* CPU with *2.66GHz* and *2GB* RAM).

Characteristics of Recommendation Settings

The performance of the algorithms is highly dependent on the recommendation problem. In this part different recommendation settings are presented as well as a discussion which algorithm performs well in these settings (an overview is given in Table 3.7). The first recommendation problem is visualized in Table 3.8. This table shows the intermediate representation (for further details on this representation see Section 3.1.2) with **n** customer requirements and **m** products. Each product does not satisfy one customer requirement. Therefore, there exists one minimal conflict set in the recommendation problem. This minimal conflict set incorporates all customer requirements. For the algorithms *GraphXplain*, *FastXplain*, *BFX* and *PFX* this

[*] All algorithms (GraphXplain, FastXplain, Boosted FastXplain, Personalized FastXplain, PersRepair, ReAction, QuickXplain, MinRelax and CorrectiveRelax) are implemented in the *D-fame: Diagnosis Framework* which is described in Chapter 5 and can be downloaded from `www.ist.tugraz.at/staff/mschuber`

Table 3.7.: Overview of the performance issues of the introduced algorithms

Recommendation Setting	Applicability of the Algorithms
low number of minimal conflict sets each with a high number of constraints	**suited** for *PersRepair* **not suited** for the algorithm *GraphXplain*, *FastXplain*, *BFX*, *PFX*, and *ReAction*
low number of minimal conflict sets each with a low number of constraints	**well suited** for all algorithms (*GraphXplain*, *FastXplain*, *BFX*, *PFX*, *PersRepair*, and *ReAction*)
high number of minimal conflict sets each with a low number of constraints	**well suited** for *GraphXplain* and *ReAction* neutral to *FastXplain*, *BFX*, *PFX*, and *PersRepair*

is worst case scenario. The algorithm *GraphXplain* builds up a fully connected graph and needs to check all fully connected sub-graphs before it finally identifies the minimal conflict set. In this recommendation problem the algorithms *FastXplain*, *BFX* and *PFX* perform equally. All of them need n iterations (n is the number of customer requirements) in order to identify the minimal conflict set. The *ReAction* algorithm also needs to build up a whole tree with the depth of n, which makes the identification of the minimal conflict set really slow. Compared to this, the *PersRepair* algorithm just needs to call the theorem prover (implemented as *QuickXplain* (Junker, 2004)) one time. The algorithm *PersRepair* is very well suited for this problem, if the theorem prover performs well.

Table 3.8.: Intermediate representation of a recommendation task with one minimal conflict set containing a high number of constraints ($MCS = \{r_1, r_2, r_3, ..., r_n\}$)

	r_1	r_2	r_3	...	r_n
p_1	0	1	1	...	1
p_2	1	0	1	...	1
p_3	1	1	0	...	1
...	...	...	...	0	...
p_m	1	1	1	1	0

Table 3.9 shows the intermediate representation of another extreme recommendation setting. This setting contains of **n** customer requirements and **m** products and one minimal conflict set. This minimal conflict set consists of only one element ($MCS = \{r_1\}$). This situation occurs if the customer has *n-1* weak requirements and one really hard one. This recommendation setting reflects the best case for the algorithms *GraphXplain*, *FastXplain*, *BFX* and *PFX*. The *GraphXplain* algorithm instantly identifies that the node r_1 is not connected to any other node and thus knows that it must be a minimal conflict set. The algorithms *FastXplain*, *BFX* and *PFX* need one iteration to identify the minimal conflict set, which makes them well

applicable in such a setting. The algorithms *PersRepair* and *ReAction* are also very well suited for this setting. For both algorithms there is no need to build up more than one level on the tree and this makes it fast.

Another recommendation setting includes only zeros in the intermediate representation. This happens when all customer requirements are over-specified. In this situation, every requirement would be a minimal conflict set. The *GraphXplain* algorithm would identify them in the first step as they are all single nodes. This is fast, which makes it applicable for this setting. The algorithms *FastXplain*, *BFX* and *PFX* build up one level of the tree in order to identify all minimal conflict sets. Compared to this, the algorithm *PersRepair* has to call the theorem prover for n times and then expand the tree with each minimal conflict set returned. On the other hand, the *ReAction* algorithm identifies the only diagnosis and builds up this one level on the tree. Based on this, it identifies that each element is a minimal conflict set. As only one call of the *ReDiagnosis* algorithm is needed, the *ReAction* algorithm is very well suited for this recommendation setting.

Table 3.9.: Intermediate representation of a recommendation problem with one minimal conflict set containing only one element ($MCS = \{r_1\}$)

	r_1	r_2	r_3	...	r_n
p_1	0	1	1	...	1
p_2	0	1	1	...	1
p_3	0	1	1	...	1
...	0	...	...	...	...
p_m	0	1	1	...	1

The recommendation settings described are all extreme settings. It is improbable that one of these settings occurs during a typical interactive recommender session. Moreover, an evaluation of the performance of the algorithms on a wider scale has been performed. Nevertheless, an understanding of the structure of the underlying problem is useful. Therefore, the *satisfaction rate* measure (see Formula 3.7) is introduced.

$$satisfactionrate(R = \{r_1, ..., r_n\}, P = \{p_1, ..., p_m\}) = \frac{1}{m*n} * \sum_{k=1}^{m} \sum_{i=1}^{n} satisfies(r_k, p_i) \quad (3.7)$$

The satisfaction rate represents the probability of a requirement being satisfied by a product (Schubert et al., 2009). A requirement is satisfied by a product if it fulfils the given customer requirement (**satisfies(r$_k$, p$_i$)**). The satisfaction rate is the normalized sum over all **n** requirements and **m** products. Using the satisfaction rate, an insight into the structure of the recommendation setting can be provided.

Although the recommendation settings presented in Table 3.8 and 3.9 have the same satisfaction rate ($\frac{m*(n-1)}{m*n} = \frac{(n-1)}{n}$), the structure of these settings is different. The minimal conflict set that can be derived from Table 3.8 incorporates many requirements, whereas the conflict set calculated from the situation presented in Table 3.9 incorporates only one requirement. Moreover, the settings shown in Table 3.8 and 3.9

are completely different from the intermediate representation containing only zero values (also the satisfaction rate is different). In a recommendation setting where the intermediate data structure contains only zero values the satisfaction rate is 0%. A recommendation problem with a high satisfaction rate has only few minimal conflict sets and diagnoses on average. On the other hand, a recommendation problem with a low satisfaction rate has more minimal conflict sets and minimal diagnoses. Nevertheless, all problem settings used, have to be inconsistent in order to apply the different algorithms for consistency management. The satisfaction rate for the product p_1 of the example introduced in Section 3.1 is: $satisfactionrate(R, p_1) = \frac{4}{6}$. In intermediate representation of the example (see Table 3.2) it is shown that the product p_1 satisfies 4 out of 6 customer requirements (R). Therefore, the satisfaction rate of p_1 and all six requirements in R is $\frac{4}{6}$. Similar to this, the satisfaction rate for the product p_2 can be calculated ($satisfactionrate(R, p_2) = \frac{2}{6}$). The overall satisfaction rate for the example is $satisfactionrate(R, P) = \frac{46}{6*15} \approx 0.51 = 51\%$.

Different Number of Product Items

A major question when dealing with recommendation problems is: *"Does the algorithm scale well on a large set of items?"*. In order to answer this question, different settings were generated that reflect typical recommendation situations. In these situations customers had to specify 15 requirements (modelled as constraints). In addition, a set of product tables with an increasing number of items (1000, 2000, ..., 10000) has been created. Finally, two datasets one with the satisfaction rate of approximately 35% and one with the satisfaction rate of approximately 50% have been created. The satisfaction rates are used to ensure that the problem is complex enough to compare the different algorithms. Moreover, two different satisfaction rates have been chosen to be able to evaluate the performance of the developed algorithms in different settings. For each evaluation setting (each algorithm, each number of product items) 100 runs have been performed to solve the over-constrained problem.

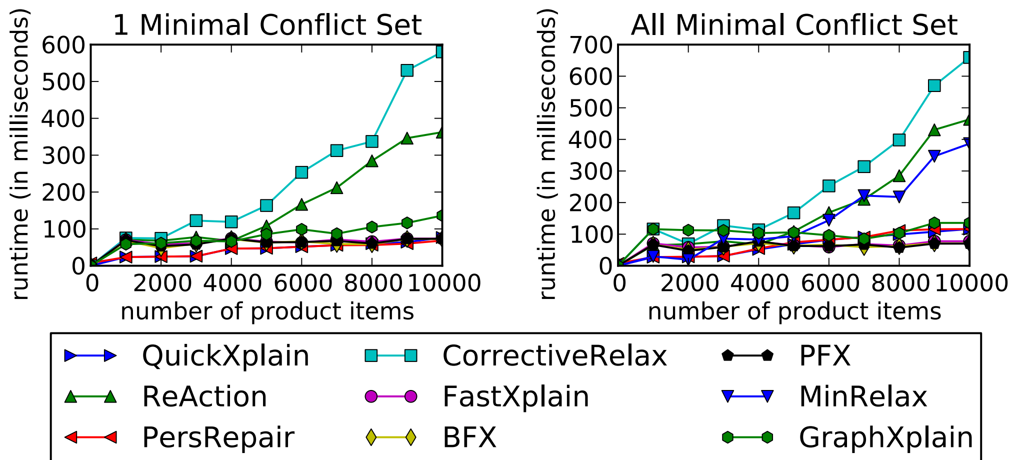


Figure 3.9.: Evaluation of all algorithms calculating one and all minimal conflict sets. The dataset contains an increasing number of items (1000, 2000, ..., 10000), 15 constraints and a satisfaction rate of 35%

Table 3.10.: Overview: how do the introduced algorithms determine **one** minimal conflict set?

Algorithm	How **one** minimal conflict set is calculated
GraphXplain	The algorithm is dedicated to calculate few or all minimal conflict sets. To calculate one minimal conflict set, the graph (one-mode network of requirements) is created and all fully connected sub-graphs are checked until one minimal conflict set has been found.
FastXplain	First, the algorithm constructs the intermediate data structure (based on the customer requirements and product assortment). Using this data structure, diagnoses are derived. A Hitting Set Directed Acyclic Graph (HSDAG) (Reiter, 1987) is constructed using these diagnoses. This is continued in a breadth-first manner, until one minimal conflict set has been identified.
Boosted FastXplain (BFX)	The algorithm calculates one minimal conflict set similar to *FastXplain*. The difference between these two algorithms is the way they derive the diagnoses, that are needed for the construction of the HSDAG.
Personalized FastXplain (PFX)	The algorithm calculates one minimal conflict set similar to *FastXplain* and *Boosted FastXplain (BFX)*. The difference between these algorithms is the way they derive the diagnoses, that are needed for the construction of the HSDAG.
PersRepair	The algorithm is dedicated to calculate few or all minimal diagnoses. These diagnoses are derived from a HSDAG constructed by minimal conflict sets (MCS). The algorithm needs to call the MCS algorithm (*QuickXplain* is used in this work) once, to calculate one minimal conflict set.
ReAction	The algorithm is dedicated to calculate few or all minimal diagnoses. In order to calculate minimal conflict sets, the algorithm calculates minimal diagnoses and builds up a HSDAG using these diagnoses. This is continued until one minimal conflict set is found.

For evaluation purposes, it has been measured how long each approach takes to calculate one minimal conflict set and all minimal conflict sets. An overview of how the introduced algorithms calculate one and all minimal conflict sets is presented in Table 3.10 (one minimal conflict set) and Table 3.11 (all minimal conflict sets). The evaluation of the algorithms calculating one and all minimal conflict sets of the dataset with a satisfaction rate of 35% is plotted in 3.9. As this figure shows, the performance of the algorithms *QuickXplain*, *PersRepair*, *FastXplain*, *BFX* and *PFX* is about the same, especially if settings with more than 5000 items are considered. The algorithm *GraphXplain* takes about the same run time for all sizes of the product sets. The run time for calculating one minimal conflict set with the *GraphXplain* is mainly used to set up the graph. The number of nodes in this graph is the same as the number of customer requirements in the recommendation problem. As for this evaluation setting the number of customer requirements is 15, the graph consists always of 15 nodes. For calculating one minimal conflict set the *ReAction* algorithm

is quite slow compared to the other algorithms. This is caused by the fact that this algorithm has to build up the whole tree. Especially, if it is compared with the *PersRepair* algorithm which can directly identify the minimal conflict set in the first iteration. Nevertheless, the *ReAction* is still competitive against the *CorrectiveRelax* algorithm (and for all diagnoses also with the algorithm *MinRelax*). Considering the right part of Figure 3.9 we see a similar picture. This is caused first by the fact, that the number of minimal conflict sets is low. And secondly by the observation that each approach needs some time to set up the data structure and this takes time. The performance of the determination of the minimal conflict sets is high, if the data structure is already instantiated. If the overall run time of all approaches is considered, then all of them are applicable for this setting (increasing number of items (1000, 2000, ..., 10000), 15 constraints and a satisfaction rate of 35% as presented in Figure 3.9) due to the fact that the highest run time (for 10000 product items) does not exceed **0.7** seconds.

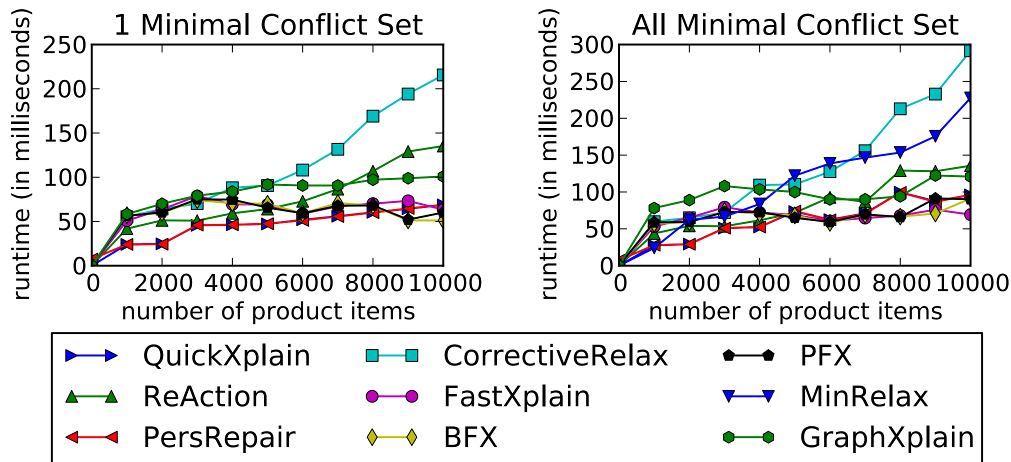


Figure 3.10.: Evaluation of all algorithms calculating one and all minimal conflict sets. The dataset contains an increasing number of items (1000, 2000, ..., 10000), 15 constraints and a satisfaction rate of 50%

Figure 3.10 shows the results of the calculation of minimal conflict sets using datasets with a satisfaction rate of 50%. The recommendation problems of this evaluation were easier to solve compared to the ones with the satisfaction rate of 35%. This is also reflected in the run time evaluation. Considering the results, the algorithms *FastXplain*, *BFX* and *PFX* need about the same time for solving the recommendation problems. In comparison, the *ReAction* algorithm performs better for less than 5000 product items, but worse for more. Nevertheless, the run times are below **0.1** second. For identifying all minimal conflict sets the algorithms *PersRepair* and *QuickXplain* have a good run time performance for less than 5000 product items. Nevertheless, these two algorithms are outperformed by the *FastXplain*, *BFX* and *PFX*. From the observations in Figure 3.9 and Figure 3.10 it can be seen that all approaches are suited for this type of settings. Nevertheless, the algorithms *PersRepair*, *FastXplain*, *GraphXplain*, *BFX* and *PFX* scale better (especially when calculating minimal conflict sets with an increasing number of items).

Table 3.11.: Overview: how do the introduced algorithms determine **all** minimal conflict sets?

Algorithm	How **all** minimal conflict sets are calculated
GraphXplain	The algorithm is dedicated to calculate few or all minimal conflict sets. To calculate minimal conflict sets, the graph (one-mode network of requirements) is created and all fully connected sub-graphs are checked to derive all minimal conflict sets.
FastXplain	First, the algorithm constructs the intermediate data structure (based on the customer requirements and product assortment). Using this data structure, diagnoses are derived. A Hitting Set Directed Acyclic Graph (HSDAG) (Reiter, 1987) is constructed using these diagnoses. This is continued until no more products are left in the leafs, because this is the indication, that all minimal conflict sets have been identified.
Boosted FastXplain (BFX)	The algorithm calculates all minimal conflict sets similar to *FastXplain*. The difference between these two algorithms is the strategy they use to derive diagnoses from the intermediate data structure. These diagnoses are needed for the construction of the HSDAG. Compared to *FastXplain*, which uses the minimality property, *BFX* uses a weight function. This difference is reflected in a different order of the diagnoses and minimal conflict sets.
Personalized FastXplain (PFX)	The algorithm calculates the minimal conflict sets similar to *FastXplain* and *Boosted FastXplain (BFX)*. Compared to *FastXplain*, which uses the minimality property, *PFX* uses a utility weight function to determine diagnoses from the intermediate data structure. This difference is reflected in a different order of the diagnoses and minimal conflict sets. The ordering of the diagnoses and conflict sets calculated by the *PFX* is strongly influenced by the utility weights.
PersRepair	The algorithm is dedicated to calculate few or all minimal diagnoses. These diagnoses are derived from a HSDAG constructed by minimal conflict sets (MCS). In order to calculate all minimal conflict sets, the algorithm needs to build a full HSDAG.
ReAction	The algorithm is dedicated to calculate few or all minimal diagnoses. For calculating all minimal conflict sets, the algorithm identifies all minimal diagnoses and builds up a HSDAG using these diagnoses.

In the following, minimal diagnoses have been calculated with all algorithms. For each algorithm 100 runs have been performed to solve the over-constrained problem. Figure 3.11 shows the run time of the algorithms for 15 customer requirements and an increasing number of product items with a satisfaction rate of 35%. Each algorithm has been stopped after it calculated one and all diagnoses (see Figure 3.11), except for the algorithm *GraphXplain*. The *GraphXplain* algorithm is dedicated to calculate minimal conflict sets. In order to calculate a diagnosis with this algorithm, the *Hitting Set Directed Acyclic Graph (HSDAG)* (Reiter, 1987) can be applied. This can be done for the calculation of one diagnosis as well as for all diagnoses. Nevertheless, the most time consuming part of the *GraphXplain* is the construction of the

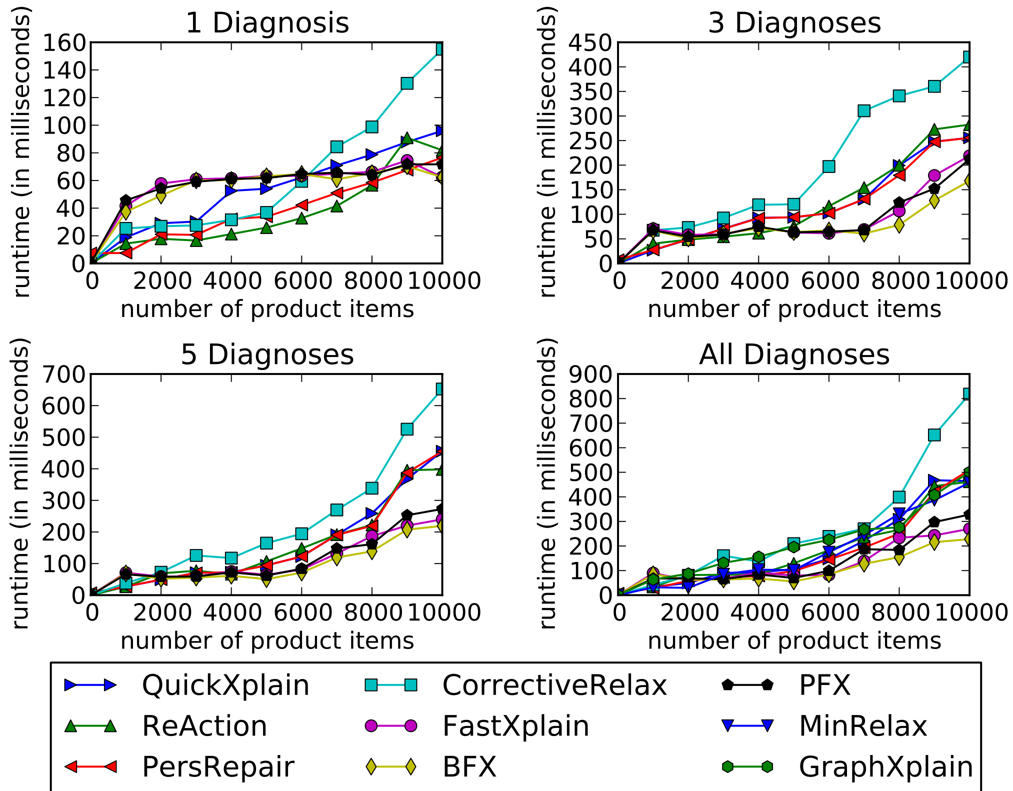


Figure 3.11.: Evaluation of all algorithms calculating one and all minimal diagnoses. The dataset contains an increasing number of items (1000, 2000, ..., 10000), 15 constraints and a satisfaction rate of 35%

one-mode network. If we would now use the *HSDAG* for calculating for one minimal diagnosis, we would have to instantiate the *GraphXplain* several times (this includes a construction of the one mode network for several times). For this reason, we decided to first calculate all minimal conflict sets with the *GraphXplain* algorithm and then build the *HSDAG*. This prevents us from several constructions of the one mode network and makes *GraphXplain* better comparable to the other algorithms. Moreover, a consequence of this decision is that the *GraphXplain* is not plotted for any settings calculating one diagnosis. An overview of how all introduced algorithms calculate one and all diagnoses is presented in Table 3.12 and 3.13.

From Figure 3.11 it can be seen that for calculating one minimal diagnosis the algorithms *PersRepair* and *ReAction* perform best over a wide range of a different number of items. The algorithms *FastXplain*, *BFX* and *PFX* have a similar run time for calculating one minimal diagnosis. When calculating more diagnoses (see for example the result of all diagnoses), the *BFX* algorithm performs slightly better compared to the algorithms *FastXplain* and *BFX*. The algorithms *PersRepair* and *QuickXplain* have similar run times. This is based on the fact that the *PersRepair* algorithm also uses the *QuickXplain* algorithm to identify minimal conflict sets. Nevertheless, the difference between these two algorithms is that *PersRepair* performs a best-first search in the hitting set directed acyclic graph (HSDAG) (Reiter, 1987) whereas our version of

QuickXplain (note that this is a combination of the HSDAG and the original *QuickXplain* (Junker, 2004)) expands the HSDAG in a breadth-first manner. The *GraphXplain* algorithm is only plotted for calculating all minimal diagnoses, for the reasons described in the last paragraph. Figure 3.11 shows that the algorithm is not the best one for calculating all diagnoses, but it is in the centre span. The *MinRelax* algorithm has been plotted only for calculating all diagnoses, because the algorithm is not dedicated to calculate only one or few diagnoses. When calculating all diagnoses, the performance of the *MinRelax* is similar to the *ReAction* algorithm. Comparing these two algorithms, it can be observed, that it is better to use the *ReAction* algorithm, because it is also suited for calculating a limited number of diagnoses. Taking a look at the *CorrectiveRelax* algorithm, it can be seen that this algorithm works well for calculating one diagnosis with settings containing less than 5000 product items. If more diagnoses are calculated, the algorithm scales badly (no matter if the number of products or the number of diagnoses is increased).

Table 3.12.: Overview: approaches to determine **one** minimal diagnosis

Algorithm	How **one** diagnosis is calculated
GraphXplain	The algorithm is dedicated to calculate few or all minimal conflict sets. The algorithm is not suited for calculating one minimal diagnosis. For this reason, *GraphXplain* was excluded from evaluations that aim to calculate one minimal diagnosis.
FastXplain	The algorithm operates on an intermediate data structure (constructed from the customer requirements and the product assortment). This data structure is used to derive diagnoses. For identifying one diagnosis, the algorithm can be stopped after deriving the first one from the intermediate data structure.
Boosted FastXplain (BFX)	Similar to *FastXplain*, the intermediate data structure is constructed and the first diagnosis is derived from this data structure. Compared to *FastXplain* a weight is used to determine the first diagnosis.
Personalized FastXplain (PFX)	See *Boosted FastXplain*.
PersRepair	The algorithm is dedicated to calculate few or all minimal diagnoses. These diagnoses are derived from a HSDAG constructed using minimal conflict sets (MCS). For calculating one minimal diagnosis, the algorithm expands the HSDAG in a best-first manner.
ReAction	The algorithm is dedicated to calculate few or all minimal diagnoses. For calculating one minimal diagnosis, the algorithm needs to be called once.

The same evaluation as described, was performed for settings with a satisfaction rate of 50%. The results can be seen in Figure 3.12. For calculating one minimal diagnosis the *PersRepair* algorithm and

Table 3.13.: Overview: approaches to determine the **complete** set of minimal diagnoses

Algorithm	How **all** diagnoses are calculated
GraphXplain	The algorithm is dedicated to calculate few or all minimal conflict sets. For calculating all minimal diagnoses, *GraphXplain* calculates all minimal conflict sets first. Using these conflict sets, a Hitting Set Directed Acyclic Graph (HSDAG) (Reiter, 1987) is constructed. From this HSDAG all minimal diagnoses can be derived.
FastXplain	The algorithm operates on an intermediate data structure (constructed from the customer requirements and the product assortment). This data structure is used to derive diagnoses. For deriving all minimal diagnoses in an intelligent way, a HSDAG is constructed. A full HSDAG indicates, that all minimal diagnoses have been found.
Boosted FastXplain (BFX)	Similar to *FastXplain*, the intermediate data structure is constructed and the diagnoses are derived from this data structure. Compared to *FastXplain* a weight is used to determine the diagnoses.
Personalized FastXplain (PFX)	Similar to *Boosted FastXplain*.
PersRepair	The algorithm is dedicated to calculate few or all minimal diagnoses. These diagnoses are derived from a HSDAG constructed using minimal conflict sets. For calculating all minimal diagnoses the algorithm needs to build up a full HSDAG.
ReAction	The algorithm is dedicated to calculate few or all minimal diagnoses. For calculating all minimal diagnoses in an intelligent way, a HSDAG is constructed. A full HSDAG indicates, that all minimal diagnoses have been found.

the *QuickXplain* perform well on the dataset. *FastXplain*, *BFX* and *PFX* scale well regarding an increasing number of product items. Although all of these algorithms need to set up the intermediate data structure before starting the calculation, they are among the better ones regarding a large amount product items. The run time performance of the algorithm *ReAction* is also acceptable for calculating one diagnosis. Figure 3.12 shows that the run time of the algorithm *CorrectiveRelax* constantly grows with an increasing number of product items. The right part of Figure 3.12 shows again, that the *GraphXplain* algorithm is among the slower ones when the dataset incorporates a low number of product items. This is caused by the fact that the algorithm needs some time to build up the graph of constraints.

Different Number of Customer Requirements

Due to the fact that the number of customer requirements may vary, the influence of the number of customer requirements on the algorithm performance must be evaluated. For a run time evaluation, several recommendation problems with 5000 product items and up to 20 customer requirements have been generated. This generation was based on a given product assortment, for which different sets of requirements have been generated in a randomized way. Two datasets have been generated, one with a satisfaction rate

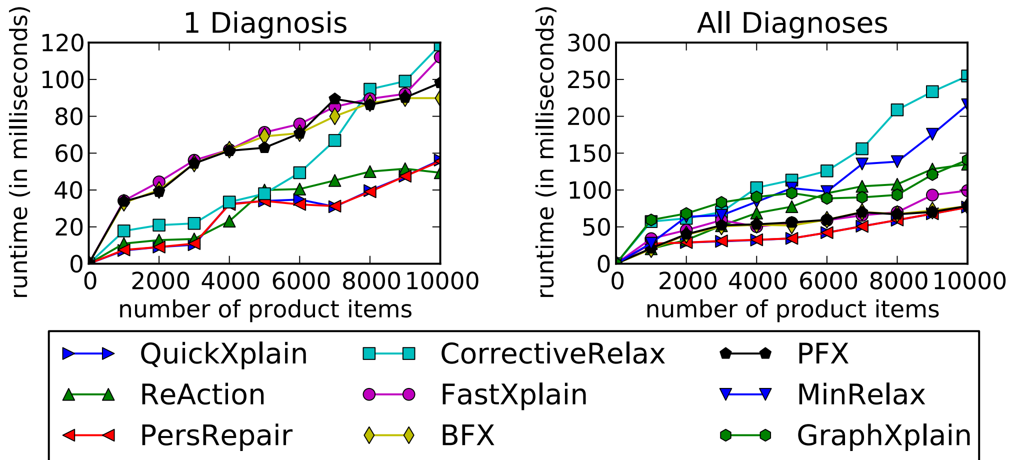


Figure 3.12.: Evaluation of all algorithms calculating one and all minimal diagnoses. The dataset contains an increasing number of items (1000, 2000, ..., 10000), 15 constraints and a satisfaction rate of 50%

of 35% and one with a satisfaction rate of 50%. These satisfaction rates have been used to ensure a realistic complexity of the problem settings. Additionally, it has been ensured that the customer requirements are inconsistent with the underlying product data. For each evaluation setting (each algorithm, each number of customer requirements) 100 runs have been performed to solve the over-constrained recommendation problem. The run times that are shown in the figures are the average values of these 100 evaluation runs.

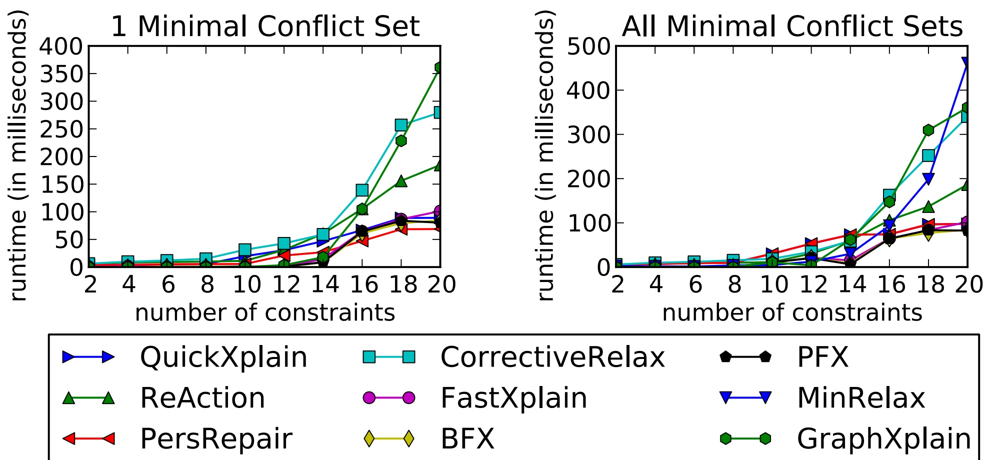


Figure 3.13.: Evaluation of all algorithms calculating one and all minimal conflict sets. The dataset contains an increasing number of constraints (2, 4, 6, ..., 20), 5000 items and a satisfaction rate of 35%

In order to compare the different approaches, it has been measured how long each approach takes to cal-

culate one and all minimal conflict sets. Figure 3.13 shows the run times in milliseconds of each algorithm for the dataset with a satisfaction rate of 35%. Comparing Figure 3.13 with another evaluation focusing on an increasing number of product items (for example, Figure 3.9), it can be observed that in the evaluations (done with an increasing number of product items) the run time increases in a linear fashion. Nevertheless, this linear progression can not be observed in the evaluations that focus on an increasing number of customer requirements.

Taking a closer look at Figure 3.13, it can be seen that up to 10 customer requirements (constraints) all algorithms are quite fast. When it comes to calculate one minimal conflict set the *CorrectiveRelax* algorithm is rather slow. Nevertheless, the situation improves when calculating all minimal conflict sets. The runtime of *GraphXplain* significantly increases with an increasing number of constraints. This is based on the fact that with an increasing number of the constraints the size of the graph increases correspondingly. Therefore, the algorithm has to elaborate on more sub-graphs which raises the run time. The algorithms *FastXplain*, *BFX* and *PFX* are all well suited for the defined recommendation problems. With these algorithms an increasing size of the intermediate data structure has a negligible impact on the runtime (see Section 3.1.2). The algorithm *PersRepair* provides the best support of one's goal is to calculate one minimal conflict set. Also, for the settings to calculate all minimal conflict sets *PersRepair* is highly recommended. The *ReAction* algorithm is again slower for calculating one minimal conflict set compared to several other algorithms. This is based on the fact that it is designed to calculate a limited set of preferred minimal diagnoses and not for calculating minimal conflict sets.

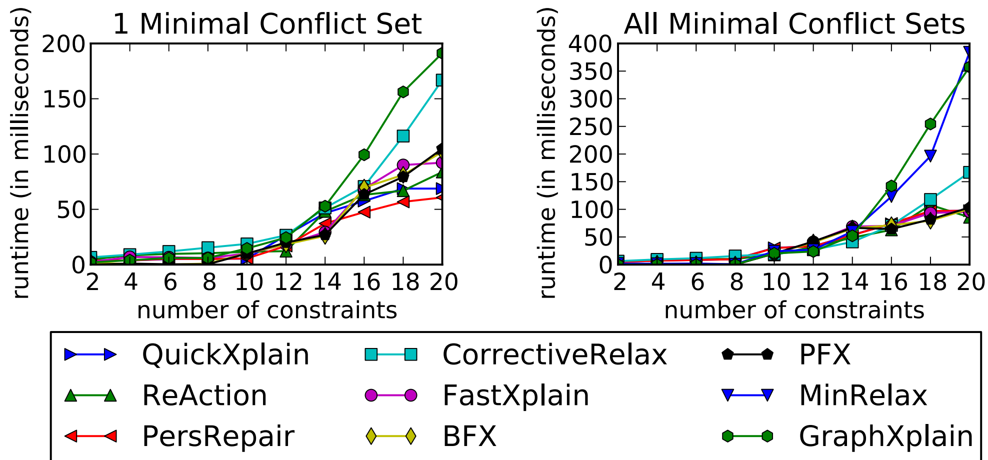


Figure 3.14.: Evaluation of all algorithms calculating one and all minimal conflict sets. The dataset contains an increasing number of constraints (2, 4, 6, ..., 20), 5000 items and a satisfaction rate of 50%

The same evaluation has been performed on the dataset with a satisfaction rate of 50%. The results of this evaluation are shown in Figure 3.14. This figure shows that the run time of the *GraphXplain* rises again very high with a higher number of customer requirements (more than 12 constraints). Considering the calculation of all minimal conflict sets (right part of Figure 3.14), it can be seen that the *GraphXplain* and the *MinRelax* algorithm have similar run times. Among the other algorithms, the *BFX* algorithm performs slightly better compared to the *FastXplain*. The *PersRepair* algorithm is again the best one to identify one

minimal conflict set for 16 to 20 customer requirements. Both Figures (Figure 3.13 and Figure 3.14) show that all algorithms stay under a run time of **0.5** seconds for calculating all minimal conflict sets.

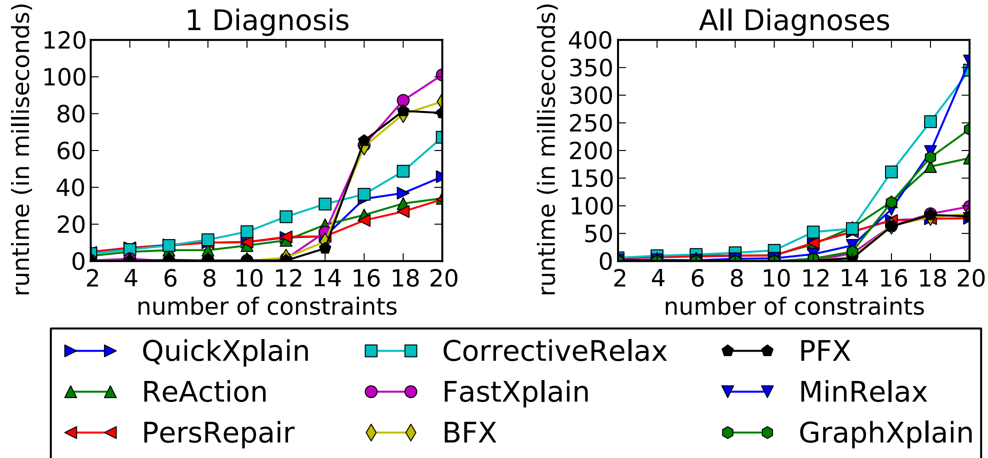


Figure 3.15.: Evaluation of all algorithms calculating one and all minimal diagnoses. The dataset contains an increasing number of constraints (2, 4, 6, ..., 20), 5000 items and a satisfaction rate of 35%

For bringing further insights into the performance of the algorithms, all of them have been evaluated with the task to calculate one and all minimal diagnoses. Figure 3.15 shows the run time of the algorithms for the dataset with a satisfaction rate of 35%. This Figure shows that the algorithms *FastXplain*, *BFX* and *PFX* perform badly when it comes to the calculation of one minimal diagnosis for a higher number of customer requirements (settings with more than 14 customer requirements). Figure 3.15 shows that the *ReAction* algorithm performs similar to the *PersRepair* algorithm for calculating one minimal diagnosis. When it comes to identifying all minimal diagnosis, the algorithms *FastXplain*, *BFX*, *PFX* and *PersRepair* perform really well for a high number of customer requirements. For a lower number of requirements (up to 14), the *GraphXplain* algorithm is also among the fastest algorithms.

The same evaluation has been performed for the dataset with a satisfaction rate of 50%. The results are shown in Figure 3.16. The evolution of the run times is similar to the ones in Figure 3.15. Nevertheless, it can be seen that the *ReAction* algorithm performs better on this dataset. It is the algorithm with the best performance when it comes to the calculation of one minimal diagnosis and it is among the best ones when it comes to the calculation of all minimal diagnoses.

Summarizing, if the run time is crucial for the recommender application there is a need to analyse the characteristics of the application before choosing an algorithm. Furthermore, issues of prediction quality play a major role in algorithm selection - this aspect will be discussed in the next section.

3.8.2. Evaluation of the Acceptance Probability (Precision)

This section aims to evaluate the algorithms regarding their acceptance probability (precision). A study performed by (Joachims et al., 2005) found out that 42% of the users clicked the top search hit, and 8% of

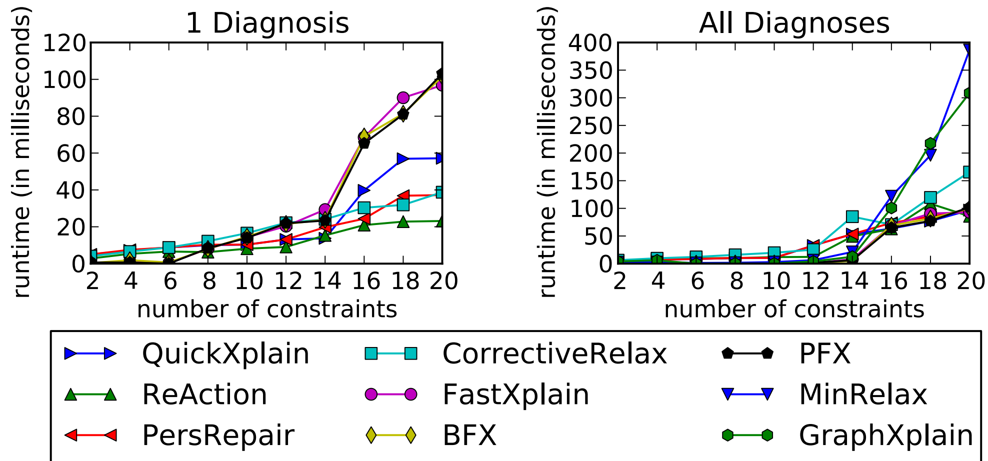


Figure 3.16.: Evaluation of all algorithms calculating one and all minimal diagnoses. The dataset contains an increasing number of constraints (2, 4, 6, ..., 20), 5000 items and a satisfaction rate of 50%

users clicked the second hit. Similar results have been identified by other studies. An important detail of this study is that the authors performed a second test in which they secretly fed the search results through a script before displaying them to users. The script swapped the two top results so that what was originally the number two entry was displayed as the number one entry and vice versa. In this situation, 34% of the users still clicked on the top entry and 12% of the users clicked the second one.

Consequently, an evaluation focusing on the acceptance probability (prediction quality) in terms of precision of the different algorithms has been performed. If the algorithm predicts the selected diagnosis on the first position then the distance is 0. The way in which such a precision is measured is defined with Formula 3.8. In this formula, **d** is the selected diagnosis and it is distinguished if this diagnosis **d** is among the top **n** (n can be chosen) items in the recommended list (i.e. precision is **1**) or not (i.e. precision is **0**).

$$precision(d,n) = \begin{cases} 1, & \text{iff } d \text{ is among the top } n, \\ 0, & \text{elsewise.} \end{cases} \tag{3.8}$$

For comparing the different approaches a user study - the *PC user study* - has been performed.

Case Study: PC User Study

For evaluating the prediction quality (precision) of the different algorithms a computer recommendation dataset has been exploited. This dataset has been composed on the basis of an online user experiment (a screen-shot of the corresponding application is shown in Figure 3.17). This experiment was conducted at Graz University of Technology. 415 subjects participated in the study (82,4% male and 17,6% female). In this study the participants had to define their requirements (R) regarding a computer. The application asked for 12 different properties of a computer. Besides the specification of their requirements, the task of the participants was to provide a ranking regarding the importance of their requirements. After the

requirement specification phase each participant was informed about the fact that no solution could be found. The system presented a list of maximal 50 different products. For each product at least one property was inconsistent with the set of requirements in R. The underlying product assortment was extracted from the *Dell* online store[†]. The ranking of the presented products was randomized. The system provided the possibility to navigate through the set using the ranking criteria *price*, *harddisk size* or *number of fulfilled requirements*. The participants had to select one computer that appeared to be the most acceptable one for them. Since no solution has been made available (only products inconsistent with R were shown), the system determined minimal conflict sets and corresponding diagnoses. The selected diagnosis is then interpreted as the set of requirements that are not satisfied by the product finally selected by the participant. The average number of diagnoses per participant was: 5.32 (std.dev. 1.67).

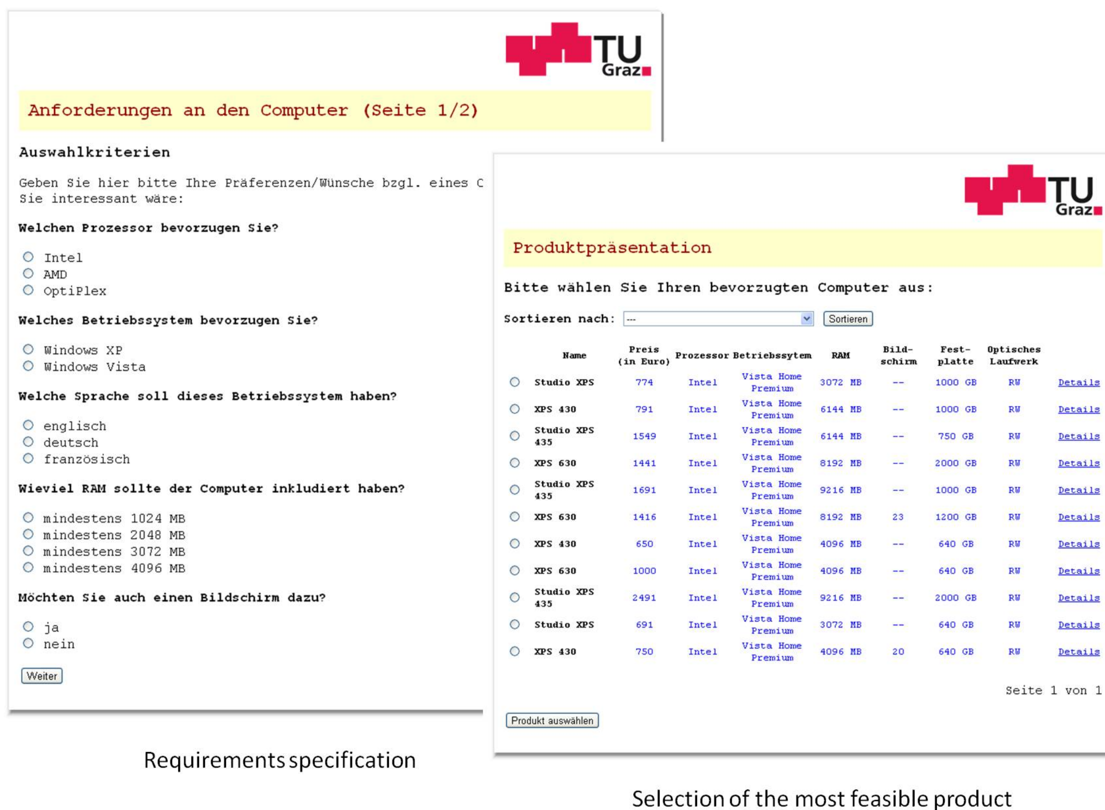


Figure 3.17.: Screen-shot of the application used for the *PC User Study*

Table 3.14 shows the results of the case study. Generally speaking, it can be observed that the precision is high in this dataset for all approaches. This is based on the fact that the average number of possible diagnoses is quite low (5.32). Another observation is that the *ReAction* and the *CorrectiveRelax* have the same precision values. The reason is that the same lexicographical ordering is used for both algorithms. The algorithms *GraphXplain*, *QuickXplain* and *MinRelax* also have similar precision values due to the fact that these algorithms sort the resulting diagnoses according to their cardinality. Among the other algorithms, the *PFX* algorithm is the best one for the first ranked diagnosis. This algorithm uses the

[†]www.dell.at

utility values specified by the user for the personalization. These utility values are also used by *ReAction*. Comparing the algorithms *PFX* and *ReAction*, it can be seen in Table 3.14 that the *PFX* has a **4%** better prediction quality compared to the *ReAction* algorithm. Although both algorithms use utility values, the different usage is responsible for this difference. Our hypothesis regarding the similarity of *ReAction* and *PersRepair* in terms of prediction quality is that this is a coincidence of the dataset.

When taking a look at the precision values for n=2 (precision = 1 if the selected diagnosis is among the first 2 diagnoses determined by the algorithm), it can be observed that the algorithms *ReAction*, *CorrectiveRelax* (O'Callaghan et al., 2005) and *PersRepair* perform well. The *PFX* algorithm is the best one regarding the top ranked diagnosis (n=1). Regarding the precision of the top two ranked diagnoses (n=2) it only has an average prediction quality. The results of the evaluation show that the algorithm *BFX* has a better performance compared to the *FastXplain*.

Table 3.14.: Average precision values of the diagnosis algorithms in the context of the *PC User Case Study*

	n=1	n=2	n=3	n=4	n=5
GraphXplain	0.51	0.75	0.87	0.92	0.98
FastXplain	0.66	0.67	0.69	0.74	0.92
Boosted FastXplain (BFX)	0.68	0.78	0.87	0.93	0.96
Personalized FastXplain (PFX)	0.74	0.82	0.87	0.95	0.99
PersRepair (similarity)	0.70	0.87	0.97	1.0	1.0
ReAction	0.70	0.88	0.97	1.0	1.0
Breadth-First (QuickXplain+HSDAG)	0.51	0.75	0.87	0.92	0.98
CorrectiveRelax	0.70	0.88	0.97	1.0	1.0
MinRelax	0.50	0.74	0.88	0.92	0.98

3.9. Related Work

The conflict detection and diagnosis approaches introduced by (Junker, 2004), (O'Callaghan et al., 2005) and (Jannach, 2008) have already been discussed. (Felfernig et al., 2004) have developed concepts to identify inconsistent customer requirements in the context of configuration problems. The idea described is to determine minimal diagnoses by applying the concepts of model-based diagnoses (Reiter, 1987). In (O'Sullivan et al., 2007) these minimal diagnoses are denoted as minimal exclusion sets. (Godfrey, 1997; McSherry, 2004) introduced approaches to identify maximally successfully sub queries which are the complement of minimal diagnoses. In order to identify minimal diagnoses, most approaches rely on the existence of minimal conflict sets. Such conflict sets can be determined, for example, on the basis of *QuickXplain* (Junker, 2004), a divide-and-conquer algorithm. The approach presented in (Felfernig et al., 2004) follows the standard breadth-first search regime for the calculation of diagnoses (Reiter, 1987). (Jannach, 2008) introduced the concept of preferred relaxations for conjunctive queries to help the customer to select a diagnosis based on a utility function.

Existing conflict detection algorithms (for example (Junker, 2004)) include an explicit consistency checking step. In general settings consistency checking is very costly and therefore should be avoided. This was our motivation for introducing the algorithms *GraphXplain*, *FastXplain*, *BFX* and *PFX*. Compared to existing conflict detection approaches (Mauss and Tatar, 2002; Junker, 2004) the introduced approaches exploit the structural properties of recommendation problems in order to avoid the consistency checks. In addition to the above mentioned ones, we developed the algorithms *PersRepair* and *ReAction* which do not rely on any structural properties of the recommendation problem and thus, can be applied for a wider range of problems.

(Schlobach et al., 2007) introduced an approach that calculates *minimal incoherent preserving* diagnoses. This work is based on the framework introduced in the earlier work (Schlobach and Cornet, 2003). The algorithm is based on a top-down method which divides the actual problem into smaller sub problems with a reduced complexity. This is followed by an informed bottom-up approach which enumerates possible solutions. In order to identify the diagnoses (Schlobach et al., 2007) uses pinpointing. These pinpoints are used to reduce a logically incorrect terminology to a smaller one. From this smaller set an error can be more easily detected by a human expert. The pinpoints prevent the algorithm from calculating minimal hitting sets by using the supersets to approximate minimal diagnoses. To compute the pinpoints themselves, all minimal conflict sets are needed, which is costly. Compared to this, the minimal conflict sets used in model-based diagnoses (Reiter, 1987) are computed on demand.

3.10. Discussion

In this chapter different algorithms for consistency management in knowledge-based recommender systems have been introduced. Knowledge-based recommender systems support customers in the identification of interesting products from large and potentially complex assortments. During the preference elicitation phase customers are repeatedly defining and revising their requirements. During this refinement, situations may occur where none of the products completely fulfils the set of requirements (Pu and Chen, 2008). In such situations the introduced algorithms can be applied.

Figure 3.18 shows a decision tree that helps to choose a consistency management algorithm. The first question asked is *"Should the algorithm be personalized?"*. Several algorithms (*PFX*, *ReAction*, *PersRepair*, *CorrectiveRelax*) have been presented that take personalisation strategies into account. The *PFX* algorithm needs a product table and utility values for the calculation. In comparison to this, the algorithms *ReAction*, *PersRepair* and *CorrectiveRelax* can operate on a product table, but they can also operate on a model representing more complex products. If a personalized algorithm is preferred, the following question is asked *"Is a product table available?"*. Depending on the answer, the algorithm *PFX* (if a table representation of the product items is available) or if it is intended to apply algorithms that rely on a lexicographical ordering of the customer requirements, the algorithms *ReAction* and *CorrectiveRelax* (O'Callaghan et al., 2005) are suggested. Otherwise, the *PersRepair* algorithm is recommended.

If there is no data available from which personalization strategies can be derived, the answer to the question *"Should the algorithm be personalized?"* is *no*. As there are more alternatives available in this context, the following question is asked: *"Is a product table available?"*. If there is no product table available, but a model of the products, it can be recommended to use a combination of the *QuickXplain* (Junker,

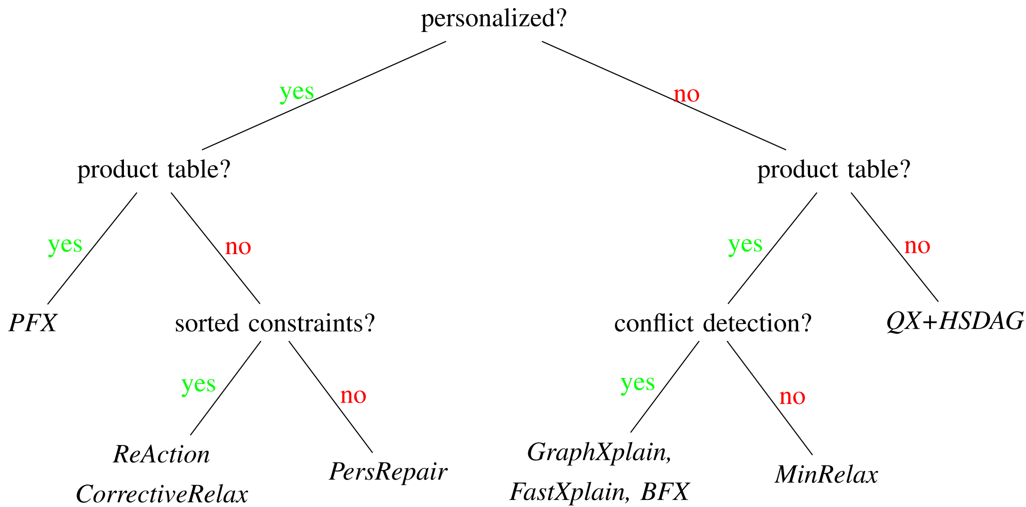


Figure 3.18.: Decision tree to decide what algorithm is suited for consistency management for constraint-based recommendation scenarios.

2004) with the hitting set directed acyclic graph (*HSDAG*) (Reiter, 1987). This combination performs a breadth-first search in the *HSDAG*. Note that the *QuickXplain+HSDAG* combination can also be applied in settings where a product table is available. If there is a product table available other algorithms can be applied as well. As there are too many algorithms that can be applied, another question is asked *"Should the algorithm focus on calculating minimal conflict sets or on minimal diagnoses?"*. Depending on the answer to the question, one of the following algorithms can be suggested: the algorithms *GraphXplain*, *FastXplain* and *BFX* (if the answer is minimal conflict sets) or the algorithm *MinRelax* (Jannach, 2008). Note that although most algorithms can calculate both (minimal conflict sets and minimal diagnoses), this distinction is made because some algorithms are more suited to calculate minimal conflict sets compared to others. Additionally, there are exceptions, for example *GraphXplain* only calculates minimal conflict sets.

Chapter 4

Consistency Management in Configuration Systems

Parts of the contents of this chapter have been published in (Felfernig and Schubert, 2010b; Felfernig et al., 2010c) (Felfernig and Schubert, 2011a; Felfernig et al., 2011).

Configuration systems have a long and successful tradition as a sub field of Artificial Intelligence (Barker et al., 1989; Mittal and Falkenhainer, 1990; Stumptner, 1997; Fleischanderl et al., 1998). On an informal level, *configuration* can be defined as a *special case of design activity, where the artefact being configured is assembled from instances of a fixed set of well-defined component types which can be composed conforming to a set of constraints* (Sabin and Weigel, 1998). This chapter focuses on *product configuration systems*. Nowadays, these systems become more and more important due to their role as a key technology of mass customization (Tiihonen et al., 2003). The goal of mass customization is to support individual customer preferences and at the same time exploit the advantages of mass production (Pine, 1999; Silveira and Fogliatto, 2001).

Configuration systems typically exploit two different types of knowledge sources: on the one hand the explicit knowledge about the *user requirements*, on the other hand a *configuration knowledge base*. Configuration knowledge bases are represented in form of a product structure and different additional types of constraints (Felfernig et al., 2004) such as *compatibility constraints* (which component types can or cannot be combined with each other), *requirements constraints* (how user requirements are related to the underlying product properties), or *resource constraints* (how many and which components have to be provided such that needed and provided resources are balanced). Knowledge-based recommender systems - in contrast to knowledge-based configuration systems - operate on a table-based product data representation. The products in a knowledge-based recommender system are explicitly represented (see example given in Section 1.1).

While interacting with a knowledge-based configuration system customers typically specify a set of requirements. These requirements are evaluated. The configurator tries to find at least one configuration which is consistent with the customer requirements and the underlying configuration knowledge base. In situations where the configurator is *not able to find a solution* customers are in the need of consistency

management support. The configurator can aid customers in consistency management using various approaches such as *FastDiag* (see Section 4.2), *PersDiag* (see Section 4.4), *CorrectiveRelax* (O'Callaghan et al., 2005), or *QuickXplain* (Junker, 2004). With the support of these algorithms customers are enabled to adapt inconsistent requirements and evaluate (new) alternative configurations (solutions).

This chapter is organized as follows: in Section 4.1 a working example from the domain of *bike* configuration is introduced. This simplified example is used throughout the chapter to illustrate the different approaches to restore consistency in configuration systems. Section 4.2 presents an approach to support customers in the consistency management (on the basis of the identification of minimal diagnoses). In the Section 4.3 this approach is adapted to identify *diagnosis clusters*. Each diagnosis cluster includes at least one minimal diagnosis, but does not need to be minimal itself. In Section 4.4 the *PersDiag* algorithm is introduced. This algorithm uses different personalization strategies in order to find diagnoses of relevance for the customer (user). In this context, the used personalization strategies are discussed and compared to each other. All technologies (introduced in Sections 4.2-4.4) help customers to solve the *no solution could be found dilemma* (Pu and Chen, 2008) in interactive configuration. A performance evaluation of the algorithms is presented in Section 4.5. Finally, the presented approaches are evaluated with regard to their prediction quality (precision). Related work is discussed in Section 4.6. In Section 4.7, this chapter is concluded with a recommendation when to use which algorithm.

4.1. Example: Configurable Bike

A simplified example of a configuration task from the domain of bike sales is introduced in this section. The example will serve as a representative problem throughout this chapter to explain the principles of the introduced consistency management algorithms. This example describes a *configuration task* (see Definition 10), which is defined as a Constraint Satisfaction Problem (CSP (Tsang, 1993)) following the work of (Felfernig et al., 2010c).

Definition 10 *A **configuration task** can be defined as a constraint satisfaction problem CSP (V, D,C) where* $V = \{v_1, v_2, ..., v_n\}$ *represents a set of domain variables,* $D = \{dom(v_1), dom(v_2), ..., dom(v_n)\}$ *is a set of domains. C is a set of constraints which can be split into* $C = C_{KB} \cup C_R$*.* $C_{KB} = \{c_1, c_2, ..., c_k\}$ *is a set of constraints that restrict the possible assignments of the variables in* V*.* $C_R = \{r_1, r_2, ..., r_i\}$ *is a set of constraints that represent the requirements of the customer.*

Relevant variables of the working example are *colour, size, gender, gear, rear, type* and *price*. The variable *colour* represents the colour of the bike, *size* is the size of the frame, *gender* is the gender type of the frame, *gear* specifies how many gears the bike has, *rear* is the brand of the rear, *type* represents the type of the bike and the *price* can be specified as well. In order to retrieve a configuration the customer can specify a value for each variable. The set of possible configurations is restricted by the constraints of the *configuration knowledge base* $C_{KB} = \{c_1, c_2, ..., c_{10}\}$. In this example the customer specifies that the type of the bike should be *city* (r_1), the frame should for *females* (r_2), the colour should be *yellow* (r_3), the bike should have more than *25* gears (r_4), the size of the bike should be *M* (r_5), a *mohawk* rear should be included (r_6) and the price should be lower than *400* (r_7). On a more formal level, the corresponding configuration task can be defined as follows:

- V = { colour, size, gender, gear, rear, type, price }
- D = {
  - dom(colour) = {*green, blue, yellow, pink*},
  - dom(size) = {*XS, S, M, L, XL*},
  - dom(gender) = {*male, female*},
  - dom(gear) = {*1, ..., 30*},
  - dom (rear) = {*shimano, mohawk, GT alloy*},
  - dom(type) = {*mtb, city, trekking, race*},
  - dom(price) = {*100, ..., 2000*}}
- C_{KB} = {
  - c_1 = {type $=$ *race* $\Rightarrow$ gender $=$ *male*},
  - c_2 = {type $=$ *city* $\Rightarrow$ gear $\leq$ *12*},
  - c_3 = {gender $=$ *female* $\Rightarrow$ size $<$ *L*},
  - c_4 = {gear $>$ *22* $\Rightarrow$ type $=$ *mtb* $\vee$ type $=$ *race*},
  - c_5 = {colour $=$ *blue* $\Rightarrow$ type $=$ *mtb* $\vee$ type $=$ *trekking*},
  - c_6 = {gender $=$ *female* $\Rightarrow$ colour $=$ *pink* $\vee$ colour $=$ *green*},
  - c_7 = {type $=$ *mtb* $\Rightarrow$ rear $=$ *GT alloy*},
  - c_8 = {type $=$ *race* $\Rightarrow$ size $\geq$ *M*}
  - c_9 = {size $=$ *M* $\Rightarrow$ colour $=$ *blue* $\vee$ colour $=$ *green*}
  - c_{10} = {gear $>$ *20* $\Rightarrow$ rear $=$ *GT alloy* $\vee$ rear $=$ *shimano* }}
- C_R = {
  - r_1 = {type $=$ *city*},
  - r_2 = {gender $=$ *female*},
  - r_3 = {colour $=$ *yellow*},
  - r_4 = {gear $>$ *25*},
  - r_5 = {size $=$ *M*},
  - r_6 = {rear $=$ *mohawk*},
  - r_7 = {price $<$ *400*}}

Based on this description of the configuration task we can introduce the definition (see Definition 11) of a concrete configuration (solution for a configuration task) based on (Felfernig et al., 2011).

Definition 11 *A* ***configuration*** *for a configuration task* (V,D,C) *is an instantiation* $I = \{v_1 = in_1, v_2 = in_2, ..., v_n = in_v\}$ *where* $in_k \in dom(v_k)$.

A configuration is *consistent* if the assignments in I are consistent with $c_i \in C$. A configuration can be denoted as *complete* if all variables in V are instantiated. Furthermore, a configuration is *valid*, if it is consistent and complete.

For the specified configuration task it is not possible to find a solution. For example, the customer specifies that the bike should be of type *city* and should have more than *25 gears*. Taking a look at the knowledge base C_{KB} it can be observed that the constraint c_2 specifies, that a *city* bike can have at most *12 gears*. Furthermore, the constraint c_6 specifies, that *female* bikes are only available in the colours *pink* and *green*. Nevertheless, the customer wants to have a *yellow* and *female* bike. Therefore, the customer requirements are inconsistent with the configuration knowledge base.

Table 4.1.: Log of the past configurations. Each log entry represents one configuration that has been designed by a user.

	colour	size	gender	gear	rear	type	price
log_1	pink	XS	female	1	shimano	city	300
log_2	green	L	male	25	GT alloy	race	800
log_3	green	M	female	10	mohawk	trekking	500
log_4	blue	M	male	12	GT alloy	mtb	400
log_5	yellow	XL	male	20	shimano	race	1100
log_6	green	S	female	16	GT alloy	mtb	500
log_7	blue	M	male	21	shimano	trekking	300
log_8	yellow	L	male	30	shimano	race	1900
log_9	green	XS	female	8	mohawk	city	800
log_{10}	pink	S	female	28	GT alloy	mtb	200

In order to identify minimal sets of requirements that have to be adapted or relaxed, the concepts of Model-Based Diagnosis (MBD) (de Kleer et al., 1992; Reiter, 1987) may be exploited. Model-Based Diagnosis starts with a description of the system. In our case this description covers the configuration knowledge base C_{KB}. The intended behaviour of the system is that it can suggest a configuration that satisfies the set of customer requirements C_R. If the actual behaviour of the system conflicts with its intended behaviour, the diagnosis component is activated. The task of this component is to identify those elements which, when assumed to be functioning abnormally, sufficiently explain the discrepancy between the actual and the intended behaviour of the system. Note that in our configuration systems it is assumed that the configuration knowledge base always functions correctly (for more information about how to debug knowledge bases see, for example, (Bakker et al., 1993; Felfernig et al., 2004; Friedrich and Shchekotykhin, 2005)). The desired diagnoses include only elements of the set of customer requirements C_R. On a more technical level a diagnosis can be seen as a minimal set of faulty components (in our case requirements) that need to be relaxed or adapted in order to be able to identify a configuration.

For our example, it is assumed that the system has already been used by customers. These customers have already successfully completed bike configurations which are stored as log entries $Log = \{log_1, log_2,$

Table 4.2.: Utility values specified by the customer (in %) during a configuration session

	type	gender	colour	gear	size	rear	price
utility	15.0	23.0	5.0	25.0	7.0	6.0	19.0

..., $log_{10}\}$. These log entries are representing complete configurations. Table 4.1 shows a list of log entries of our example. The information stored in the log can be exploited and used for personalization strategies (see also Section 4.4). For some personalization strategies it is important that the customer specifies a utility value for each attribute or requirement. For our example the utility values defined by the customer are shown in Table 4.2. A utility value indicates how important the attribute is for the customer. In our example these values are specified in % (sum of all utility values is 100%).

4.2. Algorithm: FastDiag

This section describes the algorithm *FastDiag*, which was published by (Felfernig and Schubert, 2010b; Felfernig et al., 2010c, 2011). During an interactive session with a configuration system customers specify their requirements. The *FastDiag* algorithm determines one minimal diagnosis with the same computational effort related to the calculation of one conflict set at a time. Some of the specified requirements are more important to the customer compared to others. If there exists a conflict situation, the customers usually prefer to keep the important requirements and to change or delete the less important ones (Junker, 2004). Therefore, the *FastDiag* algorithm supports the identification of preferred (leading) diagnoses based on predefined preferences. The algorithm can be applied in different scenarios such as online configuration (Felfernig et al., 2004) and recommendation (Schubert et al., 2011) as well as in scenarios where the efficient calculation of preferred diagnoses is crucial (de Kleer, 1990). In addition to constraint-based systems, the algorithm can be applied for example, in the context of SAT solving (Silva and Sakallah, 1996) and description logics reasoning (Friedrich and Shchekotykhin, 2005).

FastDiag is based the definition of a total (lexicographical) ordering as preference criteria. Using such a total ordering the customer requirements in C (especially C_R) can be sorted. This ordering can be achieved, for example, by *directly asking* the customer regarding the preferences. Another possibility to retrieve the utility values is that they are added by an expert. An expert (i.e. marketing or sales expert) specifies, for example, scoring rules or item utilities. These utility values can be used by applying multi attribute utility theory (MAUT) (von Winterfeldt and Edwards, 1986; Ardissono et al., 2003). Another possibility is to apply rankings determined by a *conjoint analysis* (Belanger, 2005). For the following discussions, the definition of a total lexicographical ordering is used (see Definition 8). For this work, the requirements are sorted according to the utilities given in Table 4.2. This results in a total lexicographical ordering of the requirements following Definition 8. The sorted requirements are $C_R = \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}$, because $r_3 < r_6 < r_5 < r_1 < r_7 < r_2 < r_4$. The most important requirement for the customer is r_4, meaning that the bike has more than *25 gears*. If we assume that $X = \{r_3, r_4, r_5\}$ and $Y = \{r_2, r_4, r_5\}$ then Y-X = $\{r_2\}$ and $X \cap \{r_4, r_5\} = Y \cap \{r_4, r_5\}$. Intuitively, $\{r_3, r_4, r_5\}$ is a preferred diagnosis compared to $\{r_2, r_4, r_5\}$ since both diagnoses include r_4 and r_5 but r_3 is less important compared to r_2. If we change the ordering to

$(r_4 < r_2 < r_7 < r_1 < r_5 < r_6 < r_3)$, *FastDiag* would determine $\{r_2, r_4, r_5\}$ as a preferred minimal diagnosis compared to $\{r_3, r_4, r_5\}$. Note that in this situation we used two diagnoses that are similar. The preferred diagnosis of the example is $d_1 = \{r_3, r_1, r_5, r_6\}$ (see also Figure 4.1).

Identification of Preferred Diagnoses

The main purpose of the *FastDiag* algorithm (see Algorithm 9) is to check, whether a diagnosis for a set of customer requirements can be calculated. The precondition for calculating a diagnosis, is that the set of customer requirements (C_R) is not empty and the knowledge base is consistent. If the whole problem is consistent the algorithm returns $\emptyset$. The check of the precondition is done by the *FastDiag* algorithm. In order to calculate a diagnosis, *FastDiag* calls the *FD* algorithm (see Algorithm 10).

The *FD* algorithm calculates a preferred diagnosis for a given set of customer requirements which are lexicographically ordered. In the last paragraph it has already been shown how the utility values of Table 4.2 (defined preferences) can be used to sort the customer requirements in a lexicographical order. The *FD* algorithm uses this sorted set ($C_R = \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}$) to calculate a preferred diagnosis. Another input parameter of the *FD* algorithm is the variable **AC**. **AC** is a set of constraints, containing the configuration knowledge base and a set of customer requirements. In the beginning this set is $AC = C_{KB} \cup C_R$, in other words it is the union of of customer requirements (C_R) and the configuration knowledge base (C_{KB}). In the working example (see Section 4.1) the customer requirements C_R are inconsistent with C_{KB} (no solution can be found that satisfies C_R and C_{KB} at the same time). This means that $C_R = \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}$ includes at least one minimal diagnosis. In the worst case the diagnosis incorporates all customer requirements C_R (i.e., each $c_i \in C_R$ represents a single conflict). This is not the case in the working example (the set of all diagnoses is $D = \{d_1 = \{r_3, r_6, r_5, r_1\}, d_2 = \{r_6, r_5, r_1, r_2\}, d_3 = \{r_3, r_5, r_4\}, d_4 = \{r_3, r_6, r_1, r_2\}, d_5 = \{r_5, r_2, r_4\}, d_6 = \{r_3, r_1, r_2, r_4\}\}$).

For calculating a preferred diagnosis, the *FD* algorithm operates on a sorted set of customer requirements ($\mathbf{C} = \mathbf{C}_R = \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}$). Additionally, a set of all constraints **AC** including the knowledge base (i.e., $\mathbf{AC} = \mathbf{C}_{KB} \cup \mathbf{C}_R$) is given to the algorithm as parameter. Moreover, the *FD* algorithm takes a delta **D**, which stores the recent changes and is emtpy in the beginning. The key idea behind the *FD* algorithm is, that a set of requirements is divided into smaller parts to narrow down the location of the minimal diagnosis.

Before calculating the diagnosis, the *FD* algorithm checks, if the set of all current constraints (**AC**) is consistent. In case of an inconsistency, the algorithm searches for these constraints that are part of the diagnosis. In the running example, the set of requirements that should be diagnosed is $C_R = \{r_3,\ r_6,\ r_5,\ r_1,\ r_7,\ r_2,\ r_4\}$. These requirements are sorted from less important requirements to more important ones. Due to the fact, that this set of requirements is inconsistent with the configuration knowledge base, it needs to be divided. Therefore, $\mathbf{C}_R = \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}$ is split up to $\mathbf{C}_1 = \{r_3, r_6, r_5, r_1\}$ and $\mathbf{C}_2 = \{r_7, r_2, r_4\}$. With respect to the lexicographical ordering, it can be observed that the requirements in $\mathbf{C}_1$ are less important to the customer. For this reason, a diagnosis consisting of (some of) these requirements may be better accepted by the customer compared to another one that includes more important requirements. Therefore, the *FD* algorithm tries to eliminate the important requirements from the diagnosis process. This is done by checking the consistency of $\mathbf{AC} = \mathbf{C}_{KB} \cup \{r_7, r_2, r_4\}$. Due to the fact that this **AC** is consistent with the configuration knowledge base, there exists at least one diagnosis in the set $\mathbf{C}_1 = \{r_3, r_6, r_5, r_1\}$. For identifying a diagnosis in this set, the *FD* algorithm is called again. In this activation, the *FD* algorithm

$FD(\emptyset, C_R = \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}, C_{KB} \cup C_R)$

$\emptyset$

r_3, r_6, r_5, r_1

$FD(\{r_3, r_6, r_5, r_1\}, \{r_7, r_2, r_4\}, C_{KB} \cup \{r_7, r_2, r_4\})$
$consistent(AC)$ ✓

$FD(\emptyset, \{r_3, r_6, r_5, r_1\}, C_{KB} \cup \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\})$
$consistent(AC)$ ×

r_5, r_1

r_3, r_6

$FD(\{r_3, r_6\}, \{r_5, r_1\}, C_{KB} \cup \{r_5, r_1, r_7, r_2, r_4\})$
$consistent(AC)$ ×

$FD(\{r_1, r_5\}, \{r_3, r_6\}, C_{KB} \cup \{r_3, r_6, r_7, r_2, r_4\})$
$consistent(AC)$ ×

r_1 r_5 r_6 r_3

$FD(\{r_5\}, \{r_1\}, C_{KB} \cup \{r_7, r_1, r_2, r_4\})$
$consistent(AC)$ ×

$FD(\{r_1\}, \{r_5\}, C_{KB} \cup \{r_5, r_7, r_2, r_4\})$
$consistent(AC)$ ×

$FD(\{r_3\}, \{r_6\}, C_{KB} \cup \{r_6, r_7, r_2, r_4\})$
$consistent(AC)$ ×

$FD(\{r_6\}, \{r_3\}, C_{KB} \cup \{r_3, r_7, r_2, r_4\})$
$consistent(AC)$ ×

Figure 4.1.: Execution tree of the algorithm *FD* to determine one preferred diagnosis. The algorithm is called with the sorted requirements $C_R = \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}$ and the knowledge base C_{KB}.

checks again if the current set of requirements ($\mathbf{AC} = \mathbf{C}_{KB} \cup \{r_3, r_6, r_5, r_1, r_7, r_2, r_4\}$) is consistent. This is not the case and for this reason, the set $\mathbf{C}_R = \{r_3, r_6, r_5, r_1\}$ needs to be further divided. The *FD* algorithm continues in the way described until either a consistent set (**AC**) is found or current set of requirements $\mathbf{C}_R$ cannot be further divided (i.e. it contains only one element). This can be observed in the right lower part of Figure 4.1 which shows the whole execution tree for the working example.

Description of the *FastDiag* Algorithm

The *FastDiag* algorithm is shown in Algorithm 9 and the *FD* algorithm is shown in Algorithm 10. In this section these two algorithms are described on a more formal level. The main purpose of the *FastDiag* algorithm (see Algorithm 9) is to check, whether a diagnosis (for all constraints in **C**) can be calculated. This is the case, if the set of constraints in **C** is not empty and the knowledge base is consistent. Otherwise (**isEmpty(C) or inconsistent(AC-C)**) the algorithm returns an empty set (**return** $\emptyset$). Note that **C** (instead of $\mathbf{C}_R$) is used as a parameter for the *FastDiag* algorithm. This can be explained by the fact that the application of the algorithm is not restricted to inconsistent sets of customer requirements.

The actual calculation of a diagnosis is done by the *FD* algorithm (see Algorithm 10). This algorithm is called with similar parameters as the *FastDiag* algorithm, namely **C**: a sorted set of constraints that should be diagnosed; and **AC**: a set of all constraints that should be considered (i.e. all constraints of **C** and some background knowledge, for example a configuration knowledge base). Additionally to these two parameters, the *FD* algorithm takes a delta **D**. In this delta the last set of changes are stored. As no changes

Algorithm 9 FastDiag(C $\subseteq$ AC, AC)

{Input: C - set of constraints that should be diagnosed (i.e. C_R)}
{Input: AC - set of all constraints (i.e. $C_{KB} \cup C_R$)}
{Output: minimal diagnosis - minimal set of faulty requirements}
if $isEmpty(C)$ or $inconsistent(AC - C)$ **then**
  return $\emptyset$
else
  return $FD(\emptyset, C, AC)$
end if

have been performed yet, the delta is empty in the beginning. Based on these parameters, the *FD* algorithm follows a divide-and-conquer strategy. This means, that the algorithm calls itself in a recursive way and in each iteration the algorithm divides the set of constraints **C** into two subsets $\mathbf{C}_1$ and $\mathbf{C}_2$.

Summarizing, the *FD* algorithm is activated with a set of constraints that should be diagnosed (**C**) and a set of all constraints that should be considered (**AC**). First, the *FD* algorithm checks, whether any change has been performed in the last iteration (i.e. the delta **D** is not empty, $\mathbf{D} \neq \emptyset$). Additionally, it is checked, if the set of all constraints *AC* is consistent (**consistent(AC)**). If this is the case, the execution of the current iteration is finished, because a diagnosis can only be calculated from an inconsistent set.

Algorithm 10 FD(D, C, AC)

{Input: D - delta set, initially empty}
{Input: C - set of constraints that should be diagnosed (i.e. $C = C_R = \{c_1, c_2, ..., c_n\}$)}
{Input: AC - set of all constraints (i.e. $C_{KB} \cup C_R$)}
{Output: minimal diagnosis - set of faulty requirements}
if $D \neq \emptyset$ **and** $consistent(AC)$ **then**
  return $\emptyset$
end if
if $singleton(C)$ **then**
  return C
end if
$k \leftarrow \lceil \frac{n}{2} \rceil$
$C_1 \leftarrow \{c_1, ..., c_k\}$
$C_2 \leftarrow \{c_{k+1}, ..., c_n\}$
$\delta_1 \leftarrow FD(C_1, C_2, AC - C_1)$
$\delta_2 \leftarrow FD(\delta_1, C_1, AC - \delta_1)$
return $(\delta_1 \cup \delta_2)$

The next check in the *FD* algorithm is, whether there is only one element in the set **C** (**singleton(C)**). As already mentioned, the *FD* algorithm performs a divide-and-conquer strategy. If a set contains only one element, then it cannot be further divided. For this reason (and because **AC** is consistent), this element is part of the diagnosis.

After the consistency and the singleton check, the set of constraints (**C**) is divided in two sets $\mathbf{C}_1$ and $\mathbf{C}_2$. The constraints in $\mathbf{C}_1$ are less important to the customer compared to the ones in $\mathbf{C}_2$ (this is based on the

lexicographical order of the constraints). The *FD* algorithm is first called with $\mathbf{C} = \mathbf{C}_2$, in order to keep as much important requirements the same as possible. The delta $\mathbf{D}$ in this first call is $\mathbf{C}_1$, because $\mathbf{C}_1$ is the complement of $\mathbf{C}_2$. The result of this call is δ_1 which can be either empty, or a set of requirements that are part of the diagnosis. Independent of the amount of elements in δ_1, this set is considered as last change (set of requirments that has been changed recently by the algorithm), that need to be taken into consideration when calling the *FD* algorithm with $\mathbf{C} = \mathbf{C}_1$. This ($FD(\delta_1, C_1, AC - \delta_1)$) is the second recursive call of the *FD* algorithm. The result of this call is again a set of constraints that are part of the diagnosis. After the subsets of $\mathbf{C}$ are considered seperatly, the *FD* algorithm needs to summarize the results. Therefore, the conjunction of δ_1 and δ_2 is returned.

Calculating More Diagnoses

Typically, a diagnosis problem has more than one diagnosis. In order to calculate more or all diagnoses the *FD* algorithm can be combined with an adapted version of the hitting set directed acyclic graph (HSDAG) (Reiter, 1987). Figure 4.2 shows the adapted graph. In this adapted HSDAG, a path is *closed*, if no further diagnoses can be identified or if the elements of the current path are a superset of an already closed path. The graph is expanded in a breadth-first manner similar to the original HSDAG in (Reiter, 1987). Applying these concepts to our working example, $\{r_1\}$ (one element of the first diagnosis ($d_1 = \{r_1, r_5, r_6, r_3\}$) from the set C_R can be deleted. Then the algorithm is restarted with $C_R - \{r_1\}$ as the set of elements that should be diagnosed. For this adapted set, another minimal diagnosis can be found. Since $AC - \{r_1\}$ is inconsistent, we can conclude that $C_R = \{r_3, r_6, r_5, r_7, r_2, r_4\}$ includes another minimal diagnosis ($d_2 = \{r_4, r_5, r_3\}$). This diagnosis is determined by the call **FD(**$\emptyset$**,** $C_R = \{r_3, r_6, r_5, r_7, r_2, r_4\}$**,** $C_{KB} \cup C_R = \{r_3, r_6, r_5, r_7, r_2, r_4\}$**)**. In order to determine all diagnoses, the algorithm has to expand all paths until each leaf is either closed or the adapted set of requirements is consistent with the knowledge base C_{KB} (i.e., a diagnosis has been identified). The adapted HSDAG for all diagnoses in our working example on the basis of *FastDiag* and *FD* is depicted in Figure 4.2.

Note that for any set of requirements (constraints in C) the algorithm *FD* always calculates the preferred minimal diagnosis in terms of Definition 8. If the diagnosis d_1 is the preferred diagnosis returned by the algorithm *FD* and one element from this diagnosis is deleted (for example $\{r_1\}$), then the next call of *FD* returns the preferred diagnosis for $C_R - \{r_1\}$. This diagnosis d_2 is less preferred compared to d_1 i.e. $d_1 >_{lex} d_2$. Consequently, diagnoses part of one path in the search tree (such as d_1 and d_2 in Figure 4.2) are in a strict preference ordering. However, there is only a *partial order* between individual diagnoses in the search tree in the sense that a diagnosis at level k is not necessarily preferable to a diagnosis at level k+1.

4.3. Algorithm: FlexDiag

Based on the algorithm *FastDiag* described in Section 4.2, the algorithm *FlexDiag* (Felfernig and Schubert, 2010b) can be exploited for identifying *diagnosis clusters*. A *diagnosis cluster* is a set of constraints that includes at least one minimal diagnosis. The *FlexDiag* algorithm allows to narrow down the diagnosis search space without the need to calculate a minimal diagnosis. This is achieved by changing the singleton property check of the *FastDiag* algorithm (see Algorithm 9). This is reflected by the statement **if**

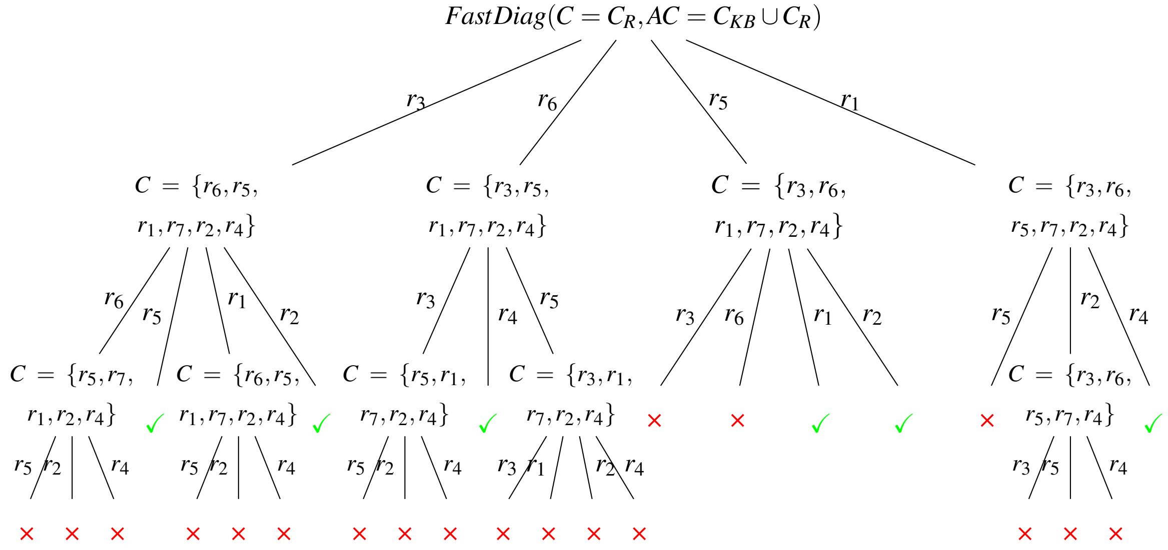


Figure 4.2.: Calculating the complete set of preferred minimal diagnoses - for this purpose, the *FastDiag* algorithm is combined with the Hitting Set Directed Acyclic Graph (HSDAG) algorithm (Reiter, 1987).

singleton(C) then (see Line 4 of Algorithm 10). This check has to be adapted to **if size(C)** $\leq$ **m**, i.e., the algorithm checks, whether the current set of constraints has m or less elements. In this way, the algorithm identifies a cluster of constraints which includes the minimal diagnosis, but often also contains elements that are not part of the minimal diagnosis. In the worst case the algorithm returns all elements of C. If m is increased in order to improve the algorithm run time, the number of consistency checks decreases.

FlexDiag leads to a set of constraints $C_D = D \cup C_{offset}$ where D is the minimal diagnosis and C_{offset} is a set of constraints which are not part of the minimal diagnosis. Note that C_{offset} can only contain constraints which are part of the requirements C_R. More precisely, C_{offset} can only contain elements that are part of **C**. An important aspect to be investigated in the *FlexDiag* algorithm is relevance. Relevance (see also Formula 4.4) is the relation between the number of relevant constraints in D and all constraints of the cluster (all constraints in C_D). For a detailed evaluation of *FlexDiag* see Section 4.5.3.

4.4. Algorithm: PersDiag

The *PersDiag* algorithm was developed to identify *personalized diagnoses of inconsistent requirements* in configuration scenarios. *PersDiag* is based on ideas of the algorithm *PersRepair* (see Section 3.6 and (Felfernig et al., 2009c)). The *PersDiag* algorithm was published in (Felfernig and Schubert, 2010a) and (Felfernig and Schubert, 2011a). This section gives insights into the algorithm and the underlying personalization strategies.

For the example described in Section 4.1 the configuration system is not able to find a solution. In order to support the customer, the approach needs to identify the minimal set of requirements $r_i \in C_R$ which have

to be adapted or relaxed so that the configuration system is able to identify at least one solution. The set of minimal conflict sets in the working example is $CS = \{CS_1 = \{r_1, r_4\}$, $CS_2 = \{r_2, r_3\}$, $CS_3 = \{r_3, r_5\}$, $CS_4 = \{r_4, r_6\}$, $CS_5 = \{r_1, r_5\}$, $CS_6 = \{r_2, r_5\}\}$. These conflict sets are minimal, i.e. there does not exist any conflict set CS'_i with $CS'_i \subset CS_i$.

The standard approach for determining minimal diagnoses, is the *hitting set directed acyclic graph* (*HS-DAG*) algorithm (Reiter, 1987). This algorithm relies on a method for determining minimal conflict sets. By resolving minimal conflicts, minimal diagnoses can be identified. Due to its minimality property, one conflict can be resolved by deleting exactly one of the elements from the conflict set. After deleting at least one element from each identified conflict set we are able to present a diagnosis. The standard *HSDAG* algorithm (Reiter, 1987) exploits the set of conflict sets in a breadth-first manner. The resolution of all minimal conflict sets leads to the identification of all minimal diagnoses. Based on the conflict sets of the working example (see Section 4.1) the following minimal diagnoses can be determined:: $Diags = \{D_1 = \{r_2, r_4, r_5\}$, $D_2 = \{r_3, r_4, r_5\}$, $D_3 = \{r_1, r_2, r_3, r_4\}$, $D_4 = \{r_1, r_2, r_3, r_6\}$, $D_5 = \{r_1, r_2, r_5, r_6\}$, $D_6 = \{r_1, r_3, r_5, r_6\}\}$.

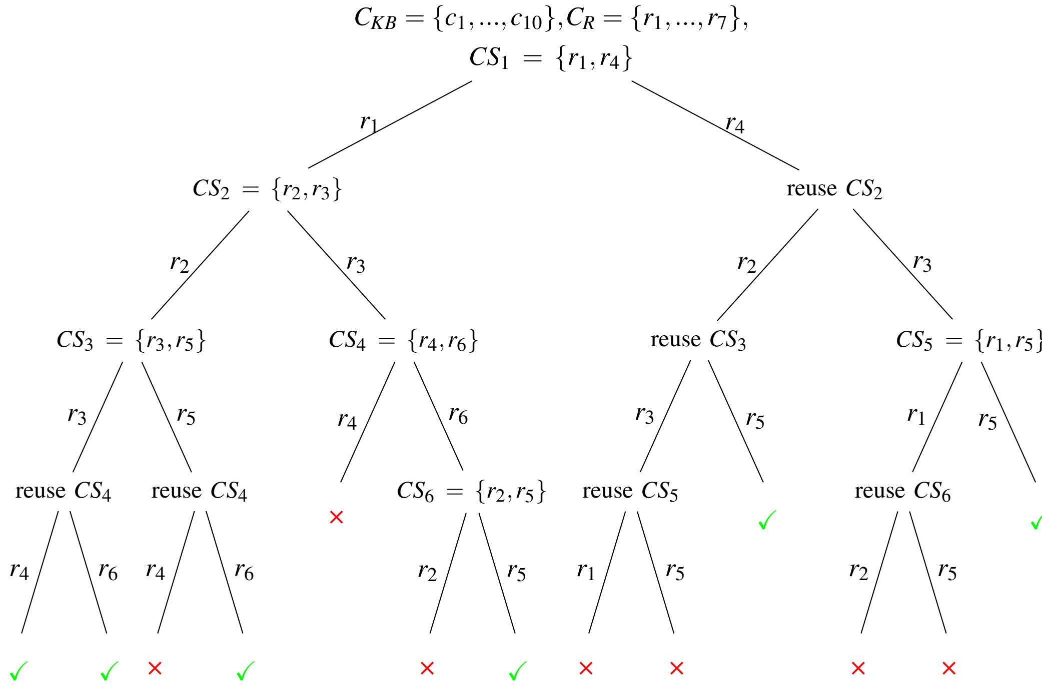


Figure 4.3.: Calculating all minimal diagnoses with the HSDAG algorithm of (Reiter, 1987)

The construction of a *HSDAG* for the example diagnosis problem is shown in Figure 4.3. The overall performance of the *HSDAG* algorithm depends on the performance of the underlying conflict detection algorithm. Conflict sets can be either calculated beforehand or during the HSDAG construction process. For our purposes, the algorithm *QuickXplain* (Junker, 2004) is used for the detection of minimal conflict sets. The *HSDAG* algorithm calls the *QuickXplain* algorithm to identify the first minimal conflict set $CS_1 = \{r_1, r_4\}$. All elements of this minimal conflict set are added as branches to the tree. Thus, for each branch it is checked, whether it already represents a diagnosis. For our example it is checked whether r_1 or r_4 is already a diagnosis. This is not the case as since ($\mathbf{C}_R$ - $\{r_1\}$) $\cup$ $\mathbf{C}_{KB}$ as well as ($\mathbf{C}_R$ - $\{r_4\}$)

$\cup$ $\mathbf{C}_{KB}$ are still inconsistent. Since both alternatives to resolve the conflict do not lead to a diagnosis, the algorithm has to further expand the tree. The *HSDAG* does this in a breadth-first manner. First, the node r_1 is expanded with the minimal conflict set $CS_2 = \{r_2, r_3\}$. This does not lead to a minimal diagnosis either and thus this conflict set can also be added to the other branch (conflict detection reuse). Note that this only works if no element of the branch is an element of the minimal conflict set. Now we can switch to the next level of the tree. The *HSDAG* algorithm inspects all nodes at level n first and moves forward to the level $n+1$. After expanding the tree with the minimal conflict sets $CS_3 = \{r_3, r_5\}$ and $CS_4 = \{r_4, r_6\}$, a minimal diagnosis $d_1 = \{r_4, r_2, r_5\}$ can be identified. This is a diagnosis because ($\mathbf{C}_R$ - $\{r_4, r_2, r_5\}$) $\cup$ $\mathbf{C}_{KB}$ does not trigger further conflicts. The *HSDAG* tree can be further expanded in order to identify all minimal diagnoses. The complete tree for our example is shown in Figure 4.3. Further details on the standard *HSDAG* (Hitting Set Directed Acyclic Graph) algorithm can be found in (Reiter, 1987; Greiner et al., 1989; Wotawa, 2001).

4.4.1. Personalization Strategies

As the number of possible diagnoses may become very large, it is important to find a way to select only the "best" (relevant) ones. A relevant diagnosis is a diagnosis that is selected by the customer, in other words, a relevant diagnosis leads the customer to an acceptable set of changes of the original requirements. One simple heuristic is to rank the diagnoses according to their cardinality. This approach has already been introduced in Section 4.4. In our example, $d_1 = \{r_4, r_2, r_5\}$ has been identified as first minimal diagnosis. Note that there exists another diagnosis ($d_2 = \{r_4, r_3, r_5\}$) as well of cardinality 3. All other minimal diagnoses have a higher cardinality. An alternative to the ranking of diagnoses based on their cardinality is to apply different types of recommendation approaches. The recommendation approaches that are investigated in this work are exploiting utility, similarity, and probability measures to guide the search for preferred diagnoses. Diagnoses with a high probability of being accepted by the customer should be ranked higher. In the following, it will be shown how the mentioned recommendation techniques can be applied to predict *relevant* diagnoses. First, we describe the different recommendation techniques. Afterwards, we show how to exploit these techniques in the diagnosis selection process.

Utility-based Recommendation

The utility-based approach is based on the *multi attribute utility theory* (MAUT) (von Winterfeldt and Edwards, 1986). This theory allows to rank different solution alternatives on the basis of a utility function. In the context of diagnosis selection the idea is to prefer diagnoses which predominantly include requirements of low importance for the customer. Following the utility-based approach of (von Winterfeldt and Edwards, 1986) we are summing up the requirement-specific importance values for each diagnosis. Based on these values we can generate a corresponding ranking.

$$utility(d \subseteq C_R) = \sum_{r_i \in d} \frac{1}{w(r_i)} * \frac{1}{\text{number of elements in d}} \tag{4.1}$$

For our example we use the utility weights introduced in Table 4.2. Based on these weights, we can calculate the utility of each diagnosis in C_R using the Formula 4.1. For the diagnosis $\{r_4, r_2, r_5\}$ the utility

value is $(1/25+1/23+1/7)*1/3=0.075$. The utility of the diagnosis $\{r_1,r_2,r_3,r_4\}$ is $(1/15+1/23+1/5+1/25)*1/4=0.087$. We finally end up with an individual importance value for each diagnosis. The utility function for a specific set **d** which is a subset of C_R is shown in Formula 4.1. In this formula, the sum of the inverted weights is multiplied by the inverted number of elements in **d**. The reason, why the number of elements in **d** is taken into account, is that we target not only the utility, but also the number of requirements that need to be changed.

Similarity-based Recommendation

The key idea of a similiarity-based diagnosis selection approach is to prefer those diagnoses which lead to configurations that resemble the original requirements most. Following this approach, the information of already existing configurations (stored, for example, in a configuration log) is exploited. It is assumed that the system has already been used by customers and that we have a log that holds all past configurations. A simplified log for our example is shown in Table 4.1. For each configuration in the log, it's similarity with the actual customer requirements can be calculated.

The determination of the similarity values is based on three attribute level similarity measures (Konstan et al., 1997; Wilson and Martinez, 1997; McSherry, 2004). The similarity is calculated for each pair of attribute a_i of the configuration log_i and the corresponding customer requirement c_i. Depending on the characteristics of the attribute one of the following three measures is taken (see also Section 3.6): *More-Is-Better* (MIB) see Formula 3.3, *Less-Is-Better* (LIB) see Formula 3.4 or *Nearer-Is-Better* (NIB) see Formula 3.5 (McSherry, 2004)

For our example we use the MIB similarity for the attribute *gear* and the NIB similarity for the attributes *colour*, *size*, *gender*, *rear* and *type*. For the attribute price we use the LIB similarity. Calculating the similarity between the attribute *gear* and the configuration log_1 (the value of *gear* is 1 in this log entry) and the customer requirement r_4 ($gear > 25$) we take the maximum $max(gear)=30$ and the minimum $min(gear)=1$ as a basis. Based on the characteristics of the attribute *gear*, we apply the MIB similarity. Therefore, the similarity of this attribute is $sim(r_4, gear) = \frac{val(r_i)-min(a_i)}{max(a_i)-min(a_i)} = \frac{1-1}{30-1} = 0$. For the MIB similarity, that is used for the attribute *gear*, applies the following: the higher the value the better it is for the customer. In comparison to this for the LIB similarity (used, for example, for the attribute *price*) applies the following: the lower the value the better it is for the customer. For a detailed discussion of different types of similarity measures see, for example, (Wilson and Martinez, 1997; McSherry, 2004).

Based on individual similarity values for each attribute we define the similarity for the whole log entry. Formula 3.6 calculates the overall similarity value between a set of customer requirements (d) and a log entry. In this context, $w(r_i)$ denotes the *importance* of requirement r_i for our example user. For our example, we assume that all requirements are equally important to the customer. Therefore, and because, the sum of all weights should be 1, the weight for each requirement r_i is $weight(r_i)=\frac{1}{7}$.

The similarity values for our working example (the customer requirements $C_R=\{r_1,...,r_7\}$ and the log entries $log_1,...,log_{10}$) are shown in Table 4.3. In this table we can already see that the log_3 has the highest similarity to the customer requirments.

Table 4.3.: Similarity between customer requirements and configuration log entries

	log_1	log_2	log_3	log_4	log_5	log_6	log_7	log_8	log_9	log_{10}
similarity	0.43	0.22	0.55	0.33	0.31	0.34	0.38	0.29	0.56	0.41

Probability-based Recommendation

The probability-based approach focuses on selection probabilities. The key idea is to suggest diagnoses to the customer that have a high probability of being selected. In order to determine the probabilities we rely on joint probabilities. For this approach, we exploit the information of already accepted diagnoses. A simplified log of accepted diagnoses for our example is shown in Table 4.4. Based on this log, we can calculate a probability value for each requirement.

Table 4.4.: Example diagnoses selected by customers. The individual probabilities are: $p(\neg r_2) = \frac{3}{5}$, $p(\neg r_3) = \frac{1}{5}$, $p(\neg r_4) = \frac{2}{5}$ and $p(\neg r_5) = \frac{3}{5}$

	colour	size	gender	gear	rear	type	price
$log-diag_1$	-	$\neq$ M	$\neq$ female	≤ 25	-	-	-
$log-diag_2$	$\neq$ green	-	$\neq$ female	-	$\neq$ GT alloy	$\neq$ race	-
$log-diag_3$	$\neq$ pink	$\neq$ M	-	-	$\neq$ shimano	$\neq$ mtb	-
$log-diag_4$	$\neq$ yellow	-	$\neq$ female	≤ 25	-	$\neq$ race	-
$log-diag_5$	$\neq$ blue	$\neq$ M	$\neq$ male	-	-	$\neq$ mtb	-

The determination of the probability $p(\neg r_1)$ ($r_1 : type = city$) denotes the probability of r_1 being part of a diagnosis. In order to calculate this probability, we are identifying the number those elements in the log (Table 4.4), where the customer selected a diagnosis containing $\neg r_1$. The probability then, is the number of such entries divided by the number of all entries in the log. Therefore, the probabilities for our example are: $p(\neg r_2) = \frac{3}{5}$, $p(\neg r_3) = \frac{1}{5}$, $p(\neg r_4) = \frac{2}{5}$ and $p(\neg r_5) = \frac{3}{5}$. If there is a requirement that is not in the log (for example r_7), we assign a small threshold to this value in order to not screw things up by a multiplication with zero. During the evaluations we observed, that a good rule of thumb to identify this value is to calculate the lowest probability in the set (in our example this is $\frac{1}{5}$) and divide it by 2. This ensures that the value is not zero, but also lower compared to the other probabilities. Note that in our example, we have three requirements that are affected by this.

Based on individual probability values for each requirement we define the probability for a set of requirements. The overall probability of a set of requirements is a joint probability. Formula 4.2 is used for determining this joint probability for a given set of customer requirements being part of a diagnosis ($C'_R \subseteq C_R$). In this formula we made the assumption of *independence of failure* which is widely applied in model-based diagnosis (de Kleer, 1990).

$$p(C'_R \subseteq C_R) = \prod_{r_i \in C_R} p(r_i) \tag{4.2}$$

Hybrid Recommendation

The main drawback of individual diagnosis prediction strategies is that they rely on one single hypothesis. The idea of hybrid diagnosis prediction strategies is to evaluate a set of hypotheses determined by individual prediction strategies for selecting the relevant diagnoses. The hybrid approach implemented within the scope of this thesis uses a simple *majority voting*. Note that there are many ways to combine algorithms to hybrids (see, for example, 2.1.4). Majority voting assumes that the errors that are made by individual predictions are not the same. Thus combining prediction strategies can be very useful for improving the prediction quality (see Section 4.5.3). A detailed study that compares different hybrid approaches is an issue for future work.

4.4.2. Identifying Personalized Diagnoses

After introducing different personalisation strategies, this section focuses on how to apply them to identify preferred (personalized) diagnoses. We use the *hitting set directed acyclic graph* (HSDAG) (Reiter, 1987) in an adapted way. Compared to the original algorithm (see Figure 4.3) we no longer use a *breadth-first* search, but a *best-first* search. The definition of *'best'* depends on the personalisation strategy.

Applying the Utility-based Strategy

Applying the utility-based selection strategy, we first call the algorithm *QuickXplain* to identify the first minimal conflict set which is $CS_1 = \{r_1, r_4\}$. The directed acyclic graph resulting from the minimal conflict sets can be seen Figure 4.4. The utility value for deleting requirement r_1 is $\frac{1}{15}$ and for deleting requirement r_4 is $\frac{1}{25}$. We are resolving this conflict set $CS_1 = \{r_1, r_4\}$ by deleting the requirement with the higher utility (r_1 in our example). The search in the tree is now continued with $C_R - r_1$. This results in the second conflict set $CS_2 = \{r_2, r_3\}$. Again we add the conflict set to the tree and calculate the utility values for the new paths ($r_1 - r_2$ and $r_1 - r_3$). We come to the conclusion that expanding the path $r_1 - r_3$ is the best choice (the utility of about 0,13 is the highest). Following this path the tree is further expanded with the conflict set $CS_3 = \{r_4, r_6\}$ and then in the same way with $CS_4 = \{r_2, r_5\}$. After calculating the utility values we see that the path $r_1 - r_3 - r_6 - r_5$ is the best choice. It would now be the turn to further expand this node. But since $C_R - \{r_1, r_3, r_6, r_5\} \cup C_{KB}$ is already consistent we found our first personalized diagnosis: $D_1 = \{r_1, r_3, r_6, r_5\}$.

Applying the Similarity-based Strategy

The idea behind the similarity-based strategy is that the customer prefers those diagnoses which lead to configurations that resemble the original requirements (configurations that are most similar to the original requirements). For the similarity-based selection of diagnoses we again assume that the *QuickXplain* algorithm (Junker, 2004) returns as first conflict set $CS_1 = \{r_1, r_4\}$. There are two possibilities of resolving this conflict set. We can either delete the requirement r_1 or the requirement r_4 from C_R. If we delete r_1, the following configurations from the log are consistent with $\neg r_1$: $Config(\neg r_1) = \{log_{2,...,8,10}\}$. We are interested in these configurations, because they are all inconsistent with the requirement r_1. We can do the same for the requirement r_4 which results in the configurations $Config(\neg r_4) = \{log_{1,...,7,9}\}$. The

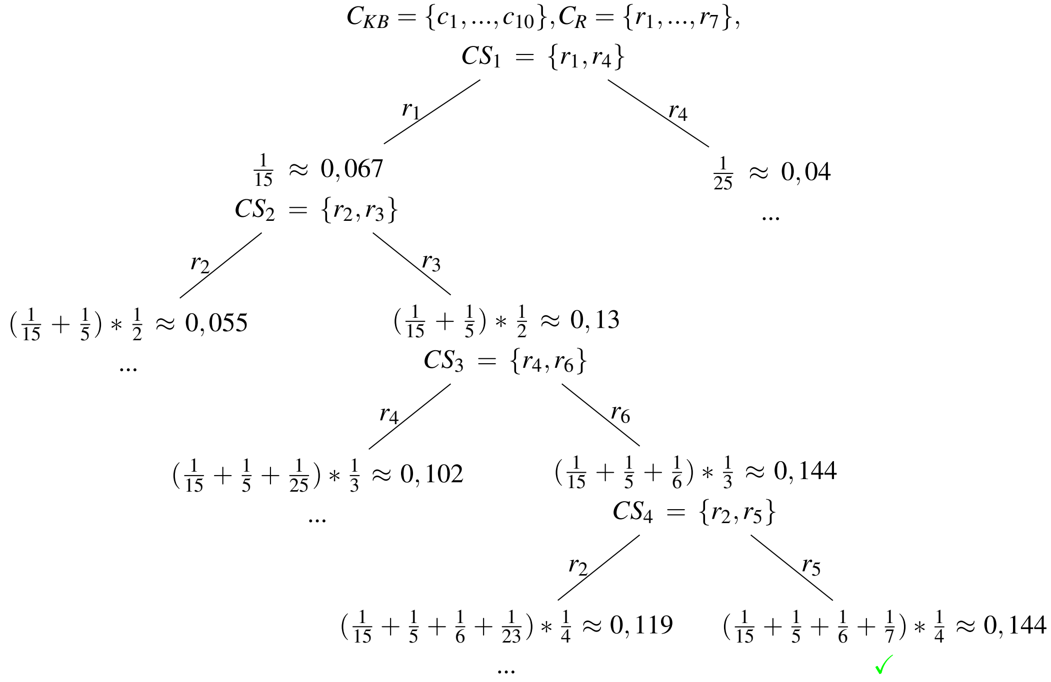


Figure 4.4.: Calculating the preferred diagnosis with the algorithm *PersDiag* using utility values. The preferred diagnosis is $\{r_1, r_3, r_6, r_5\}$

configuration with the highest similarity of the log is log_9 (see Table 4.3). This log entry is contained in $Config(\neg r_4) = \{log_{1,...,7,9}\}$. As we are performing a best-first search in the tree, we are expanding the tree into the direction with the configuration that has the highest similarity compared to the original requirements C_R. Since $C_R - r_4$ is still inconsistent with C_{KB}, we are expanding the tree into the direction of r_4 (see also Figure 4.5). The next conflict set that is returned by the *QuickXplain* algorithm (Junker, 2004) is $CS_2 = \{r_2, r_3\}$. We are again resolving this conflict set and calculating all configurations of the log that are not satisfying the requirements of the path. This results in $Config(\neg r_4, \neg r_2) = \{log_{2,4,5,7}\}$ and $Config(\neg r_4, \neg r_3) = \{log_{1,2,3,4,6,7,9}\}$. We are again expanding the tree into the direction of the configuration with the highest similarity, namely into the direction of r_3. This leads us to the next minimal conflict set: $CS_3 = \{r_1, r_5\}$. After another identification of the configurations from the log we further check the path $r_4 - r_3 - r_5$. As $C_R - \{r_4, r_3, r_5\}$ is consistent with the configuration knowledge base C_{KB}, we have finally found a diagnosis. This preferred diagnosis is $D_1 = \{r_4, r_3, r_5\}$.

If the system is used over a longer period of time, the number of configurations in the log may become very large. In order to reduce the set of configurations that need to be considered, a nearest neighbour approach may be used (similar to the approach introduced in Section 3.6). Another problem that occurs in many configuration scenarios is the *ramp-up problem* (Burke, 2000). If it is a configuration system that can configure a lot of diverse configurations, there is no configuration data available that is similar to the customer requirements. An approach to deal with this, is to define a threshold value which specifies an upper similarity limit for configurations to be accepted as similar to the original set of requirements. If no configuration exists that lies above this threshold, a fall-back solution is to present diagnoses resulting from breadth-first search or to apply the utility strategy.

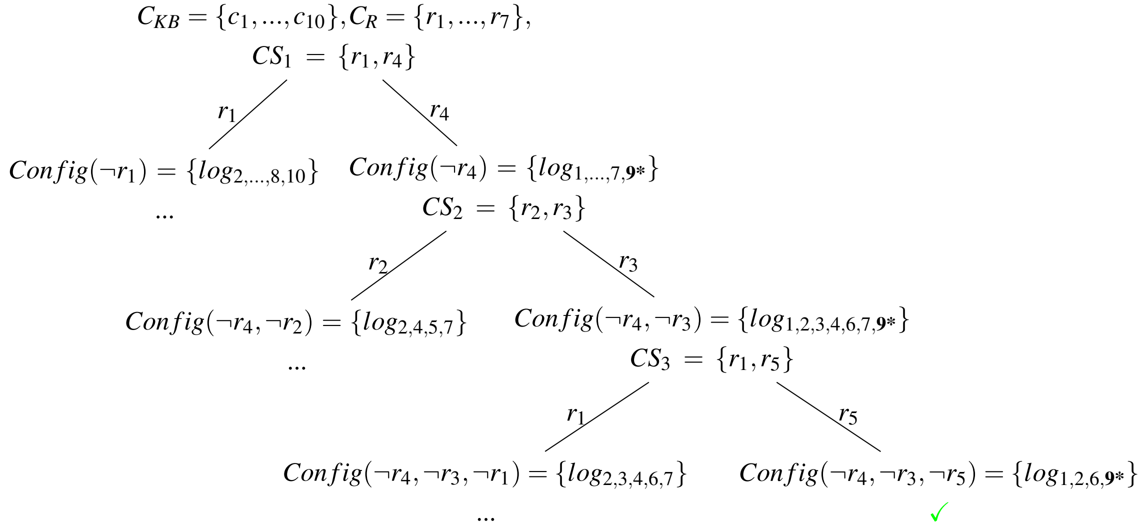


Figure 4.5.: Calculating the preferred diagnosis with the algorithm *PersDiag* using similarity values. The preferred diagnosis is $\{r_4, r_3, r_5\}$

Applying the Probability-based Strategy

For the probability-based selection of diagnoses, we again assume that the *QuickXplain* algorithm (Junker, 2004) returns as first conflict set $CS_1 = \{r_1, r_4\}$. There are two possibilities of resolving this conflict set. We can either delete the requirement r_1, or the requirement r_4 from C_R. Thus, we are identifying the probability values for $p(\neg r_1)$ and $p(\neg r_4)$. Based on the diagnoses log (see Table 4.4) we can easily calculate the probability for r_4: $p(\neg r_4) = \frac{2}{5}$. For the requirement r_1 there is no entry in the log. Consequently, we are applying the threshold value of $p(\neg r_1) = 0.1$. If we would not apply this threshold, the probability value would be $p(\neg r_1) = 0$. In this case it would be infeasible to expand the tree into the direction of r_1.

Based on the probabilities of r_1 and r_4 we can see that it is more likely that the customer will change r_4 compared to r_1. Thus we are expanding the tree into the direction of r_4. The complete tree to identify the preferred diagnosis is shown in Figure 4.6. The next minimal conflict set that is returned is $CS_2 = \{r_2, r_3\}$. We have again two possibilities to resolve this conflict. For the paths $r_4 - r_2$ the probability is: $p(\neg r_4, \neg r_2) = \frac{2}{5} * \frac{3}{5} = 0.24$ and for the path $r_4 - r_3$ it is: $p(\neg r_4, \neg r_3) = \frac{2}{5} * \frac{1}{5} = 0.08$. The node resulting from the path $r_4 - r_2$ is the one with the highest probability, thus we are expanding the graph in this direction. For the current situation ($C_R - \{r_4, r_2\}$) we are still able to find a minimal conflict set, namely $CS_3 = \{r_3, r_5\}$. Again we update the probabilities of the new nodes and try to further expand the path $r_4 - r_2 - r_5$. As $C_R - \{r_4, r_2, r_5\}$ is already consistent with the configuration knowledge base C_{KB}, we finally found a diagnosis. This preferred diagnosis is $D_1 = \{r_4, r_2, r_5\}$ with an acceptance probability (precision) of $14,4\%$.

If the system is used over a longer period of time, it is more likely that all requirements are part of the diagnoses log. Therefore, the need for the threshold is reduced.

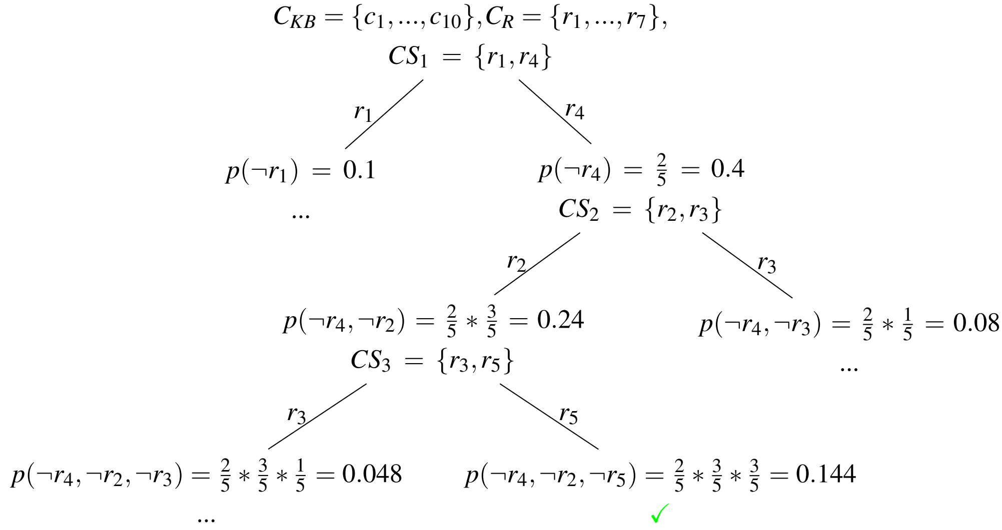


Figure 4.6.: Calculating the preferred diagnosis with the algorithm *PersDiag* using probability values. The preferred diagnosis is $\{r_4, r_2, r_5\}$

Description of the *PersDiag* Algorithm

The number of possible diagnoses can become very large. Confronting the customer with such a large number of alternatives is inappropriate. Moreover, we want to systematically reduce the number of alternatives. The goal is to identify relevant diagnoses for the customer and to keep the diagnosis evaluation process as simple as possible. We already introduced different personalisation strategies that can be used for this purpose. Now the focus lies on a more formal description of how to identify personalized diagnoses.

Algorithm 11 PersDiag(C_R, C_{KB}, H, k)

```
{Input: C_R - set of user requirements}
{Input: C_KB - the configuration knowledge base}
{Input: H - collection of all paths in the search tree (initially empty)}
{Input: k - k most similar items to be used by PersonalisedSort(H,k)}
{Output: d - preferred diagnosis}
d ← first(H)
CS ← TP((C_R − d) ∪ C_KB)
if CS = ∅ then
   return d
end if
for all requirements r from CS do
   H ← H ∪ {d ∪ {r}}
end for
H ← PersonalisedSort(H,k)
PersDiag(C_R, C_KB, H, k)
```

The idea is to apply a best-first search strategy using the concepts of the hitting set directed acyclic graph

(HSDAG) (Reiter, 1987). *PersDiag* is outlined in Algorithm 11. As the algorithm is based on the HSDAG (Reiter, 1987), we keep the same level of description for this algorithm. The *PersDiag* algorithm takes the customer requirements $\mathbf{C}_R$, the configuration knowledge base $\mathbf{C}_{KB}$, an empty collection (**H**) for all paths in the search tree and the sorting criteria **k** as input values. The collection **H** stores all paths of the search tree in a best-first fashion. The currently best path **d** is the one with the most promising (partial) diagnosis. The theorem prover ($\mathbf{TP((C_R\text{-}d)} \cup \mathbf{C}_{KB})$) is called with the set of current requirements (depending on the path **d**) and the configuration knowledge base $\mathbf{C}_{KB}$). The task of the theorem prover is to calculate the next minimal conflict set. If the current set of requirements $\mathbf{C}_R\text{-}\mathbf{d}$ is consistent with the knowledge base, the theorem prover returns an empty conflict set. In this case ($\mathbf{CS} = \emptyset$) we found a diagnosis. In the case that the theorem prover found a minimal conflict set, all elements of this set are added to the collection **H**. After the new elements have been inserted, we finally sort all paths according to the personalization criteria **k**. In this context, **k** represents the criteria used for selecting the next node to be expanded in the search tree which could be *breadth-first*, *similarity-based*, *utility-based* or *probability-based*. In order to calculate more then one diagnosis, the *PersDiag* algorithm is called in a recursive way, starting again with the most promising path.

4.5. Evaluation

In this section we present an evaluation and comparison of the introduced algorithms with state-of-the-art approaches. The evaluation is divided into three parts, namely the *performance analysis* (Section 4.5.1), *performance evaluation* (Section 4.5.2) and the *prediction accuracy evaluation* (Section 4.5.3). In the *performance analysis* we are analysing the different techniques according to their run time behaviour. The focus lies especially on the discussion of the worst case and the best case. In the *performance evaluation* we are evaluating the run time performance of the different approaches by executing them on real problems. For the performance evaluation we used the *CLib: Configuration Benchmarks Library*[*] as well as an industrial dataset. Our experiments have shown that the run time is mainly influenced by the number of customer requirements as well as by the number and structure of the constraints in the configuration knowledge base. Therefore, this thesis focuses on presenting the evaluation outcome of problems with an increasing number of customer requirements (constraints). In the *prediction accuracy evaluation*, the prediction quality in terms of precision is evaluated.

In order to compare our algorithms with state-of-the-art approaches, we implemented the following two additional algorithms. The first one is a combination of the *QuickXplain* (Junker, 2004) and the hitting set directed acyclic graph (*HSDAG*) (Reiter, 1987). This combination was also described in Section 4.4 and from now on is referred to as *QuickXplain*. This algorithm performs a breadth-first search in the *HSDAG* and uses the *QuickXplain* as a theorem prover. Note that we also implemented the pruning functionality (Greiner et al., 1989; Wotawa, 2001) into our *HSDAG*. The second approach we used for comparison purposes is the *CorrectiveRelax* algorithm by (O'Callaghan et al., 2005). This algorithm aims to calculate *corrective explanations*. A corrective explanation is the complement of a diagnosis (relaxation). The basic algorithm just identifies one corrective explanation (and the corresponding diagnosis), but we can integrate this basic approach with the *HSDAG* approach (Reiter, 1987) calculating all corrective explanations.

[*] www.itu.dk/research/cla/externals/clib

4.5.1. Performance Analysis

This section aims to compare the algorithms *FastDiag* (introduced in Section 4.2) and *PersDiag* (introduced in Section 4.4) with the *QuickXplain* algorithm (see above) and the *CorrectiveRelax* algorithm (O'Callaghan et al., 2005).

The worst case complexity of *FastDiag* in terms of number of consistency checks needed for the calculation of one minimal diagnosis is $2\mathbf{d}\cdot\log_2(\frac{n}{d})+2\mathbf{d}$, where **d** is the set size of the minimal diagnosis and **n** represents the number of constraints (in C). The best case complexity is $\log_2(\frac{n}{d})+2\mathbf{d}$. In the worst case, each element of the diagnosis is contained in a different path of the search tree: $\log_2(\frac{n}{d})$ is the depth of the path, 2**d** represents the branching factor and the number of leaf-node consistency checks. In the best case, all elements of the diagnosis are contained in one path of the search tree.

The worst case complexity of *QuickXplain* in terms of consistency checks needed for calculating one minimal conflict set is $2\mathbf{k}\cdot\log_2(\frac{n}{k})+2\mathbf{k}$, where **k** is the minimal conflict set size and **n** is again the number of constraints (in C) (Junker, 2004). The best case complexity of *QuickXplain* in terms of the number of consistency checks needed is $\log_2(\frac{n}{k})+2\mathbf{k}$ (Junker, 2004). The algorithm *PersDiag* uses the *QuickXplain* algorithm to determine the minimal conflict sets.

Consequently, the number of consistency checks per conflict set (*QuickXplain*) and the number of consistency checks per diagnosis (*FastDiag*) have similar complexity. Let $\mathbf{n}_{cs}$ be the number of minimal conflict sets in C (C_R) and $\mathbf{n}_{diag}$ be the number of minimal diagnoses, then the *FastDiag* algorithm needs $\mathbf{n}_{diag}$ calls and $\mathbf{n}_{cs}$ additional consistency checks to determine *all minimal diagnoses*. Furthermore, $\mathbf{n}_{cs}$ activations of *QuickXplain* with n_{diag} additional consistency checks are needed for determining *all minimal diagnoses* with the standard HSDAG-based approach (Reiter, 1987). The *PersDiag* algorithm also needs $\mathbf{n}_{cs}$ activations of *QuickXplain* with n_{diag} additional consistency checks to determine *all minimal diagnoses*.

In addition we compare the introduced approaches to the *CorrectiveRelax* algorithm (O'Callaghan et al., 2005), which allows to calculate minimal preferred diagnoses by constructing the complement of a maximal relaxation. The difference between *FastDiag* and *CorrectiveRelax* is the following: We first generate a minimal diagnosis D_1 in the set of constraints C_1 for the set of constraints $AC - C_2$. Next a minimal diagnosis D_2 in the set C_2 is generated taking the constraints in $AC - D_1$ into account. Both minimal diagnoses are combined with the final minimal diagnosis. Broadly speaking, *CorrectiveRelax* (O'Callaghan et al., 2005) would search in the whole set $AC - D_1$ for generating a minimal diagnosis D_2, which leads to unnecessary consistency checks. For a more detailed discussion of *CorrectiveRelax* we want to refer to (O'Callaghan et al., 2005).

4.5.2. Performance Evaluation

In this Section the performance evaluation is presented. All algorithms[†] are implemented in Java 1.6 and all experiments have been executed on an desktop PC (*Intel®Core*™2 *Quad CPU Q9400* CPU with *2.66GHz* and *2GB* RAM).

For the run time performance evaluation we randomly generated constraints that customer requirements which are inconsistent with the underlying configuration knowledge base. The Table 4.6 we give an

[†] All algorithms (FastDiag, FlexDiag, PersDiag as well as the algorithms QuickXplain and CorrectiveRelax) used for the evaluation, are implemented in the *D-fame - Diagnoses Framework* which is described in Chapter 5

overview of the different evaluation settings. For the evaluation we calculated one ($n = 1$ diagnosis), few ($n = 5$ and $n = 10$ diagnoses) or all diagnoses.

Table 4.5.: Overview: how to determine the **complete** set of minimal diagnoses?

Algorithm	How **all** diagnoses are calculated
FastDiag	The algorithm is dedicated to calculate minimal diagnoses. One minimal diagnosis is calculated by activating the algorithm once. For calculating all minimal diagnoses in an intelligent way, a HSDAG is constructed.
FlexDiag	Similar to *FastDiag*, but the calculated diagnoses are not necessarily minimal
PersDiag	The algorithm is dedicated to calculate minimal diagnoses. Diagnoses are derived on the basis of a HSDAG. For calculating one minimal diagnosis, one or more minimal conflict sets are needed.

Table 4.6.: Overview of the configuration knowledge bases that are used for the run time performance tests

	source	#Variables	#Rules
FS	www.itu.dk (CLib Benchmark)	23	16
PC2 (referred as PC)	www.itu.dk (CLib Benchmark)	40	19
Bike2 (referred as Bike)	www.itu.dk (CLib Benchmark)	34	32
Renault	www.itu.dk (CLib Benchmark)	101	113
Financial Service	www.hypo-alpe-adria.at	25	15
Broadband	www.variantum.com (WeCoTin tool)	242	84

CLib Configuration Benchmarks

For the first case study the *CLib: Configuration Benchmarks Library*[‡] has been used. Out of this benchmark library four different settings with different complexity have been selected. The financial service (*FS*) setting (see Table 4.6) is the smallest one with 23 variables and 16 rules (see also the overview in Table 4.6). Compared to this the *Renault* is the biggest setting from this benchmark library with 101 variables and 113 rules. The results of the run time performance evaluation can be seen in Figures 4.7-4.10. We decided the evaluate the number of consistency checks (not the run time in *milliseconds*). The reason for this is that the run time can be influenced by other applications or processes running on the same computer, whereas the consistency checks are independent from other processes.

[‡] www.itu.dk/research/cla/externals/clib

The consistency checks are performed with CLab (Jensen, 2004). CLab[§] is an open source C++ library for building fast backtrack-free interactive product configurators (Hadzic et al., 2004). The product configuration is realized in two phases. In the first phase a Binary Decision Diagram (BDD) (Bryant, 1986) is constructed that holds the information of valid configurations. We decided to keep this process offline. In the second phase, this BDD is accessed by the online interactive product configurator (Hadzic et al., 2004). For the evaluation all knowledge bases have been compiled into the BDDs.

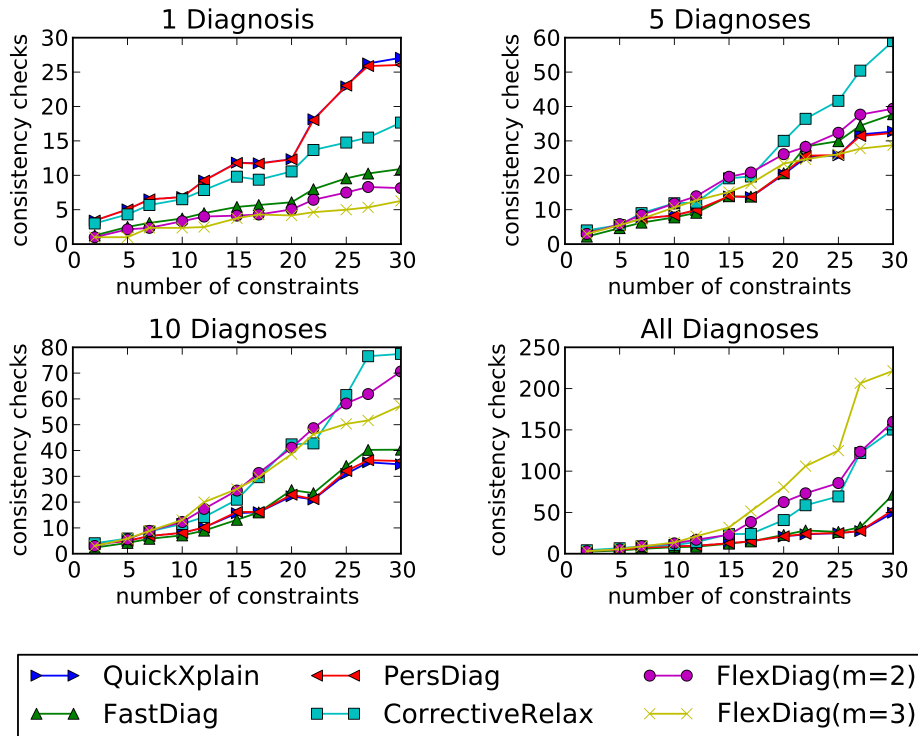


Figure 4.7.: Performance evaluation of the *CLib Benchmark FS* to calculate 1, 5, 10 and all minimal diagnoses with an increasing number of constraints

The first setting analysed, is the *CLib Benchmark FS* dataset. The dataset incorporates only 16 rules. Figure 4.7 shows the results of the evaluation. For the evaluation, settings with an increasing number of requirements have been generated. For each setting, each approach calculated 1, 5, 10 and all diagnoses. For each number of constraints 100 settings have been generated (100 times a different set of requirements that is inconsistent with the configuration knowledge base). For calculating one diagnosis the *FlexDiag (m=3)* is the approach with the best run time performance. Nevertheless, it only calculates a cluster which contains at least one diagnosis. The *PersDiag* algorithm is the slowest one for calculating one minimal diagnosis because it needs to build up a hitting set graph with the minimal conflict sets. The approach with the best run time performance (on average over all settings) that calculates minimal diagnoses, is the *FastDiag*. The number of consistency checks that are needed for the calculation of five diagnoses are similar for all approaches. In order to calculate ten diagnoses, the algorithms *PersDiag* and *FastDiag* need a similar amount of consistency checks similar to the combination of the *QuickXplain* (Junker, 2004) and

[§] www.itu.dk/people/rmj/clab/

the *HSDAG* (Reiter, 1987). For calculating all minimal diagnoses, it can be observed that *FlexDiag* needs more consistency checks. This is based on the fact that *FlexDiag* does not calculate minimal diagnoses and thus has to check more nodes in the tree in order to identify all diagnoses clusters.

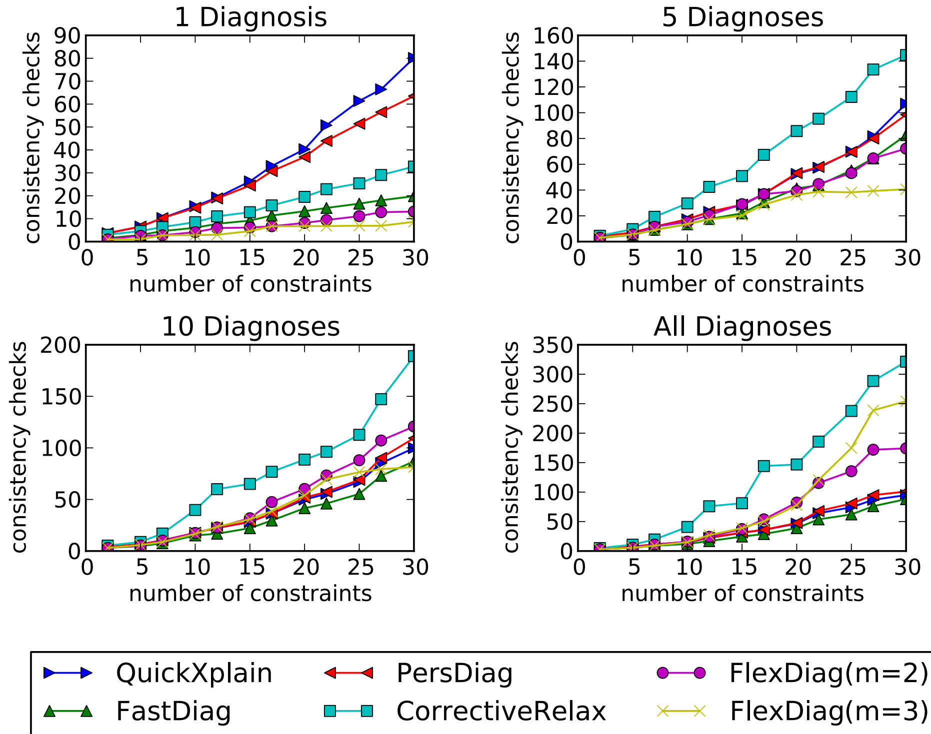


Figure 4.8.: Performance evaluation of the *CLib Benchmark PC* to calculate 1, 5, 10 and all minimal diagnoses with an increasing number of constraints

The next setting is the *CLib Benchmark PC* dataset. The dataset incorporates only 19 rules and 40 variables. Figure 4.8 shows the results of the evaluation. For the evaluation, constellations with an increasing number of requirements have been generated. For each setting, each approach was called with the task to calculate 1, 5, 10 and all diagnoses. For each evaluation (containing from 2 up to 30 requirements) 100 settings (100 times a different set of requirements that is inconsistent with the configuration knowledge base) have been generated. The *QuickXplain* is the slowest one for calculating one diagnosis (n=1), because it needs most steps to identify one diagnosis with breadth-first search in the *HSDAG*. The algorithm *FlexDiag (m=3)* needs the lowest number of consistency checks. Nevertheless, that approach does not calculate a minimal diagnosis. What can be seen in the left upper part of Figure 4.8 is that the algorithms *FastDiag*, *FlexDiag (m=2)* and *FlexDiag (m=3)* have a similar performance. Summarizing, when we are looking for an approach to determine only one diagnosis, the following question has to be answered: is it enough to just calculate a diagnosis cluster or is a minimal diagnosis needed? This cannot be answered on a general basis, but is dependent on the domain and the application. For calculating 5 diagnoses *FlexDiag (m=3)* is still the one that needs the lowest number of consistency checks. This changes when calculating 10 diagnoses. In this setting the *FastDiag* is the one that performs best. Moreover, it can be observed that for calculating 5 and 10 diagnoses the algorithms *PersDiag* and *QuickXplain* perform similarly. For

calculating all diagnoses the *FlexDiag (m=3)* is better compared to the *CorrectiveRelax*.

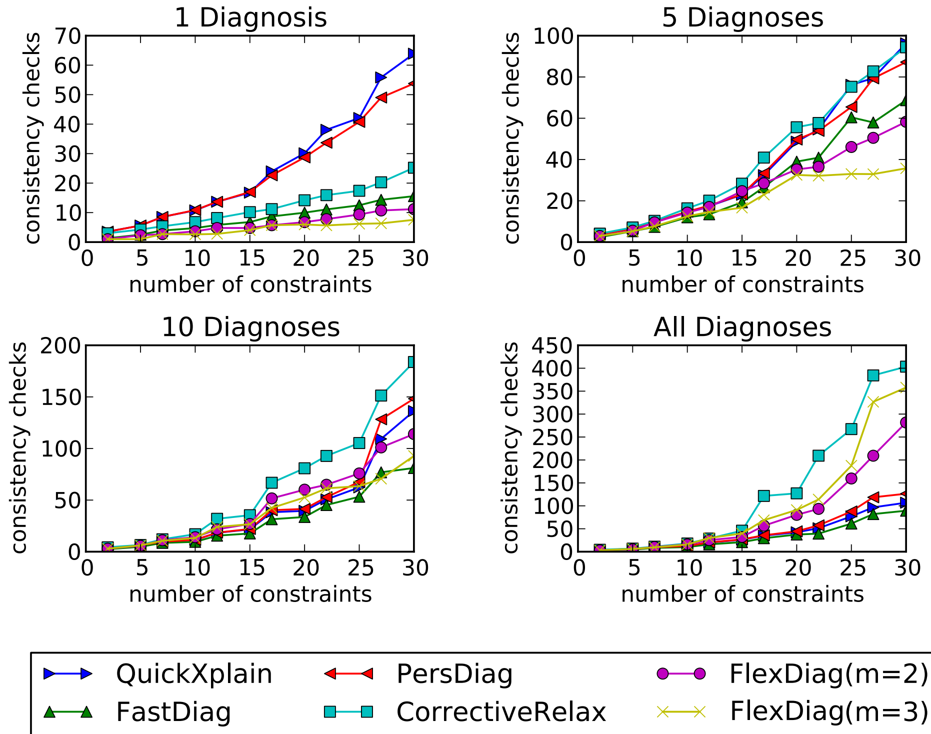


Figure 4.9.: Performance evaluation of the *CLib Benchmark Bike* to calculate 1, 5, 10 and all minimal diagnoses with an increasing number of constraints

The introduced approaches have also been evaluated on the *CLib Benchmark Bike* dataset. This dataset has twice as many rules as the CLib Benchmark *FS* dataset. Nevertheless, the results are similar as shown in Figure 4.9. When comparing the algorithms *PersDiag* and *QuickXplain*, it can be observed that for calculating one diagnosis the *PersDiag* needs less consistency checks due to the best-first search strategy. For calculating 10 diagnoses the tendency of all approaches is similar. The results for the determination of all diagnoses are similar to the ones of the *CLib Benchmark PC* dataset.

The *CLib Benchmark Renault* dataset is the most complex setting of the benchmark library. It consists of 113 rules and 101 variables. The result of the evaluation is shown in Figure 4.10. Using a high number of constraints (requirements) the algorithms *FastDiag*, *FlexDiag (m=2)*, *FlexDiag (m=3)* and *CorrectiveRelax* scale much better compared to the algorithms *PersDiag* and *QuickXplain*. The results for determining five diagnoses are similar to the results of the calculation of one diagnosis. Another observation of the evaluation is that the algorithms *FastDiag* and *CorrectiveRelax* have a similar performance. Compared to this in the settings where one diagnosis and ten diagnoses are calculated, the *FastDiag* performs better compared to the *CorrectiveRelax* algorithm. For determining all diagnoses, the *FastDiag* is the approach that needs the lowest number of consistency checks. *QuickXplain* performs better for calculating all diagnoses compared to *PersDiag*. This is based on the fact that *QuickXplain* uses a breadth-first search strategy. Nevertheless, the diagnoses calculated by *PersDiag* are personalized and can be seen as more valuable to

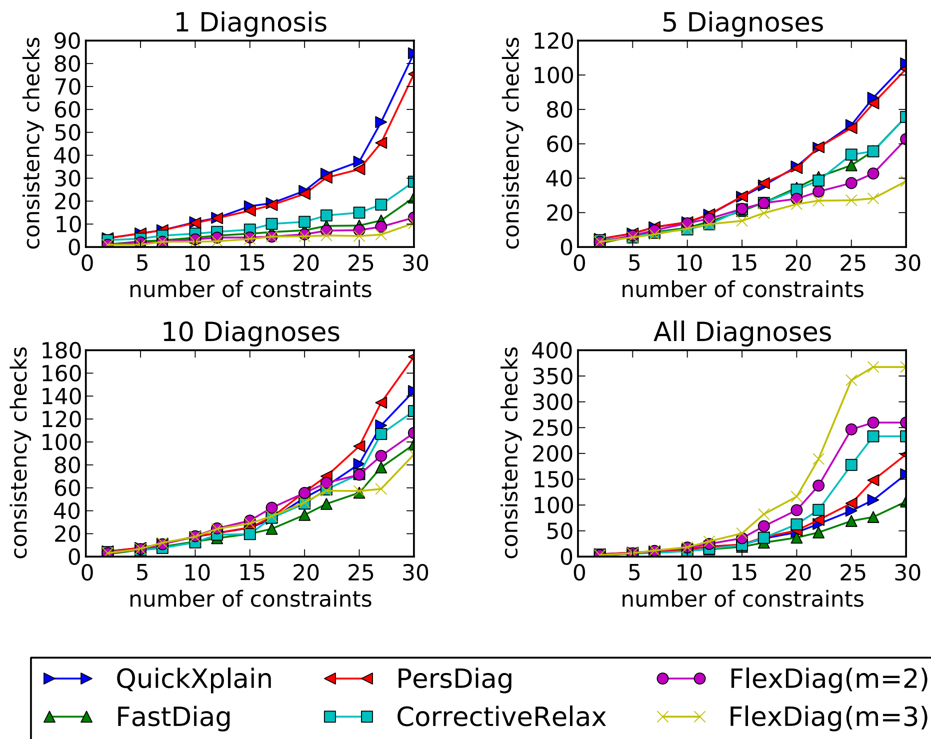


Figure 4.10.: Performance evaluation of the *CLib Benchmark Renault* to calculate 1, 5, 10 and all minimal diagnoses with an increasing number of constraints

the customer.

WeCoTin Models

This section presents the results of the evaluation performed with *SModels* inference engine. This engine consists of a front-end (*lparse*) and inference procedure *SModels* (see (Tiihonen et al., 2003) for a detailed description). The settings of this evaluation were modelled within the *WeCoTin* project (Tiihonen et al., 2003). This project established a platform for web configuration technology. The *WeCoTin* system (Tiihonen et al., 2003) is available through a standard browser and configuration knowledge bases are available on a configuration server. The evaluations presented here were performed directly with the configuration server. One interesting finding of the evaluation is, that the evaluation results of the *WeCoTin* Models are similar to the ones performed with the *CLib Configuration Library*. The result of the evaluation performed with the biggest *WeCoTin* model can be seen 4.11. The domain of this setting is the telecommunication domain. Figure 4.11 shows the number of consistency checks that were needed for calculating 1, 5, 10 and all diagnoses for an increasing number of constraints.

Figure 4.11 shows that the algorithms *PersDiag* and *QuickXplain* (Junker, 2004) scale badly for a high number of constraints when determining one diagnosis. The reason for this is that both approaches need to build the *HSDAG* (Reiter, 1987) with the minimal conflict sets. This needs more time compared to

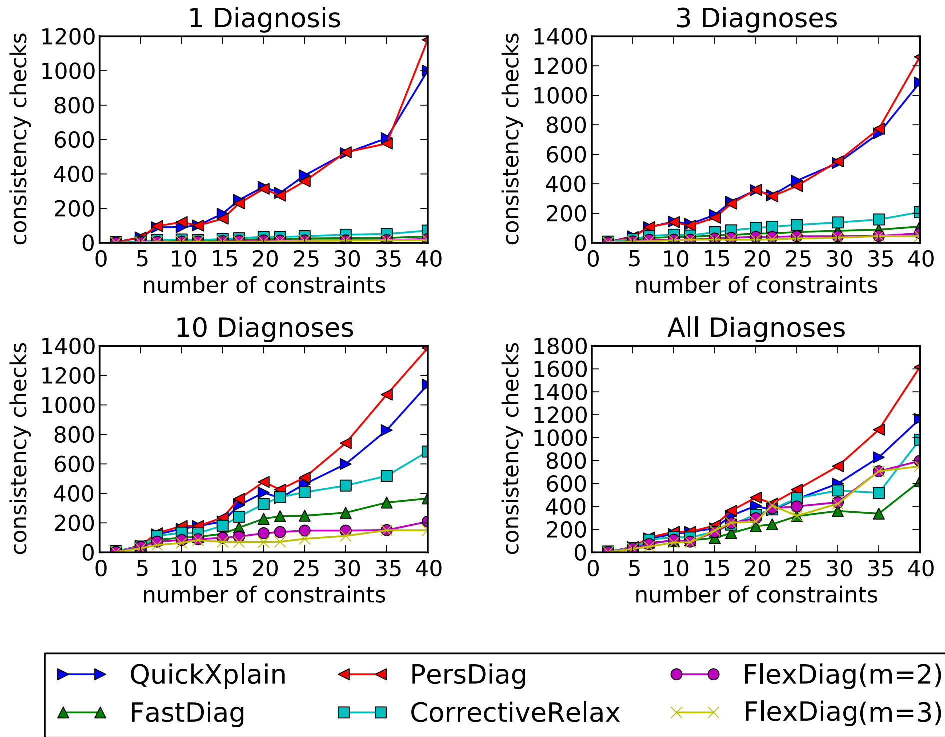


Figure 4.11.: Performance evaluation of the *broadband* model to calculate 1, 5, 10 and all minimal diagnoses with an increasing number of constraints (from 2 to 40 constraints)

a direct calculation of the diagnoses. Figure 4.12 shows the same evaluation but with a lower number of constraints. In this figure it can be seen that for calculating all diagnoses for less than 12 constraints *FlexDiag (m=3)* is the one that needs the lowest number of consistency checks. For a higher number of constraints *FastDiag* is the one that needs the lowest number of consistency checks. Moreover, it can be observed that for identifying all diagnoses - especially minimal diagnoses - *FastDiag* is suited best for more than 12 constraints.

As shown in Figure 4.11, the algorithms *PersDiag* and *QuickXplain* need a similar number of consistency checks for calculating 1 and 3 diagnoses. For calculating more diagnoses (10 or all) the *PersDiag* needs more consistency checks due to the best-first search strategy.

In the Appendix of this thesis more evaluation results with models of the *WeCoTin* project are provided. These tables show the number of needed solver calls for calculating one and all diagnoses for 10 and 20 customer requirements. The results are similar to the ones just described.

4.5.3. Evaluation of the Acceptance Probability (Precision)

This section aims to evaluate the different diagnosis approaches with regard to their acceptance probability (precision). For evaluating the acceptance probability, two case studies have been used. In each case study the difference between the position predicted by the algorithm and the expected position (position 1), was

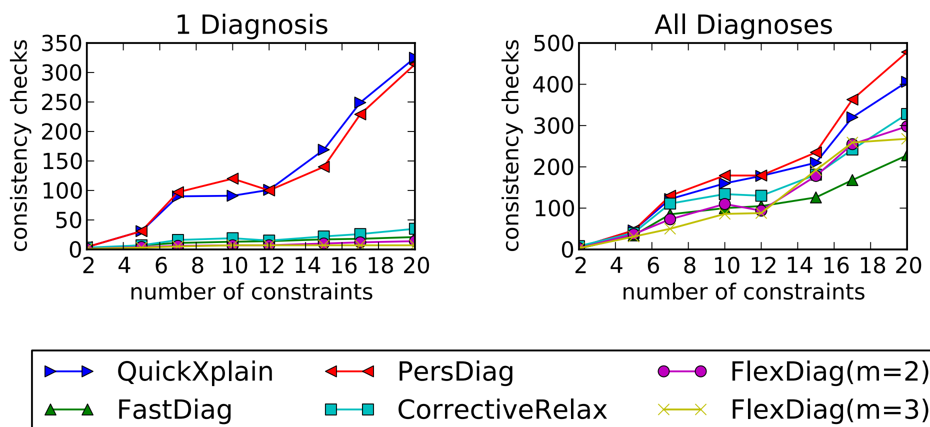


Figure 4.12.: Performance evaluation of the *broadband* model to calculate 1, 3, 10 and all minimal diagnoses with an increasing number of constraints

analysed. If the diagnosis algorithm predicts the selected diagnosis on the first position this distance is 0. The precision measure is shown in Formula 3.8 (see page 77). Precision describes how often a diagnosis that corresponds to a *diagnosis selected by the customer* is among the top-n ranked diagnoses. One case study (*PC User Study*) was performed at the Graz University of Technology. For the second case study a dataset of one of the largest financial service providers in Austria was used.

Case Study: PC User Study

In one case study we used a computer configuration dataset (the dataset introduced in Chapter 3) for evaluating the prediction quality of the presented diagnosis algorithms. In this study we used a configuration knowledge base (CSP-based representation) in contrast to the product table based representation used in Chapter 3.

Table 4.7 shows the results from the *PC User Study*. The precision is high due to the low number of diagnoses (see also Chapter 3). Another observation is that *FastDiag* and *CorrectiveRelax* (O'Callaghan et al., 2005) have the same precision. The reason is that both algorithms are using the same lexicographical ordering.

The algorithm with the best prediction quality (precision) for $n = 1$ is *PersDiag (utility)*. This algorithm uses the utility values specified by the user. These utility values are also used by *FastDiag*, although in a slightly different way. Compared to*FastDiag*, it can be seen that *PersDiag (utility)* has a **4%** better performance. This performance gap is caused by the different usage. As described in Section 4.2 the *FastDiag* algorithm uses the utility values to sort the requirements before starting the calculation of the diagnosis. Compared to this, the *PersDiag (utility)* (see Section 4.4) retrieves one minimal conflict set at a time. After a minimal conflict set has been added to the Hitting Set Directed Acyclic Graph (HSDAG) (Reiter, 1987), the utility values of all leafs are re-evaluated. Taking a look at the precision for the first two rankings, it can be seen that the precision is approximately the same (**0.88** for the *FastDiag* and **0.89** for the *PersDiag (utility)*).

Table 4.7.: Precision values of the algorithms for the *PC User Case Study*

	n=1	n=2	n=3	n=4	n=5
FastDiag	0.70	0.88	0.97	1.0	1.0
PersDiag using Utility	0.74	0.89	0.96	1.0	1.0
PersDiag using Similarity	0.70	0.87	0.97	1.0	1.0
PersDiag using Probability	0.65	0.78	0.87	0.94	0.97
PersDiag using Hybrid-Approach	0.72	0.85	0.90	0.96	1.0
Breadth-First (QuickXplain+HSDAG)	0.51	0.75	0.87	0.92	0.98
CorrectiveRelax	0.70	0.88	0.97	1.0	1.0

PersDiag (Hybrid) is a combination of *PersDiag (utility)*, *PersDiag (similarity)* and *PersDiag (probability)*. The performance of this algorithm lies between the best algorithm (*PersDiag (utility)*) of these three and the worst (*PersDiag (probability)*). The hybrid approach aims to combine different strategies to overcome the drawbacks of individual strategies. However, with the Computer dataset no further improvements compared to the individual approaches in terms of prediction quality could be observed.

Case Study: Financial Services

Another case study that has been conducted for evaluating the acceptance probability (precision) of the presented diagnosis algorithms, is based on a dataset of a financial service configuration system. This system was developed for one of the largest financial service providers in Austria. The interaction log incorporates 1703 sessions and includes information about which attributes values have been specified by the customer (including the order of the requirements). In 418 sessions the specified requirements became inconsistent with the underlying knowledge base. For those sessions the dataset also holds the information about which diagnosis has been selected by the customer in order to restore consistency. The average number of diagnoses presented to the customer was 20.42 (std.dev. 4.51).

In order to show the prediction quality of different approaches introduced in this chapter, the interaction log of a financial service configurator application has been used. For this case study, precision quality (see Formula 3.8) of the different approaches has been measured. Table 4.8 shows the result of the evaluation. The *Breadth-First* approach - a combination of the algorithms *QuickXplain* (Junker, 2004) and the *HSDAG* (Reiter, 1987) - has the lowest precision. This approach ranks the diagnoses according to their cardinality and can be used as a base line. The algorithms *FastDiag* and *Corrective Relax* (O'Callaghan et al., 2005) have the same precision due to the same requirements ordering strategy. In Table 4.8, it can be seen that *FastDiag* performs better compared to *PersDiag (utility)*. Although both algorithms use utility values, the different usage is responsible for this performance gap. An important point when comparing the results of the *PC user study* (see Table 4.7) and the *Financial Service* (see Table 4.8) case study is that the overall precision of the second case study is much lower. The reason for this is, that the precision depends not only on the approaches used, but also on the dataset. Especially the average number of diagnoses (*PC user study*: **5.32**, *Financial Service*: **20.42**) affects the overall precision of the approaches.

Table 4.8.: Precision values of the algorithms for the *Financial Service Case Study*

	n=1	n=2	n=3	n=4	n=5	n=7	n=10
FastDiag	0.179	0.368	0.534	0.640	0.700	0.820	0.935
PersDiag using Utility	0.175	0.361	0.500	0.569	0.625	0.745	0.893
PersDiag using Similarity	0.166	0.370	0.481	0.583	0.648	0.740	0.865
PersDiag using Probability	0.148	0.328	0.467	0.564	0.634	0.768	0.879
PersDiag using Hybrid-Approach	0.172	0.331	0.499	0.582	0.631	0.753	0.895
Breadth-First (QuickXplain+HSDAG)	0.133	0.290	0.391	0.529	0.626	0.691	0.870
CorrectiveRelax	0.179	0.368	0.534	0.640	0.700	0.820	0.935

In order to gain further insights into the prediction quality of the presented diagnosis algorithms a two-sample T-test has been performed. The aim was to figure out whether there exists a significant difference between the diagnosis methods in terms of their *mean absolute error* (MAE).

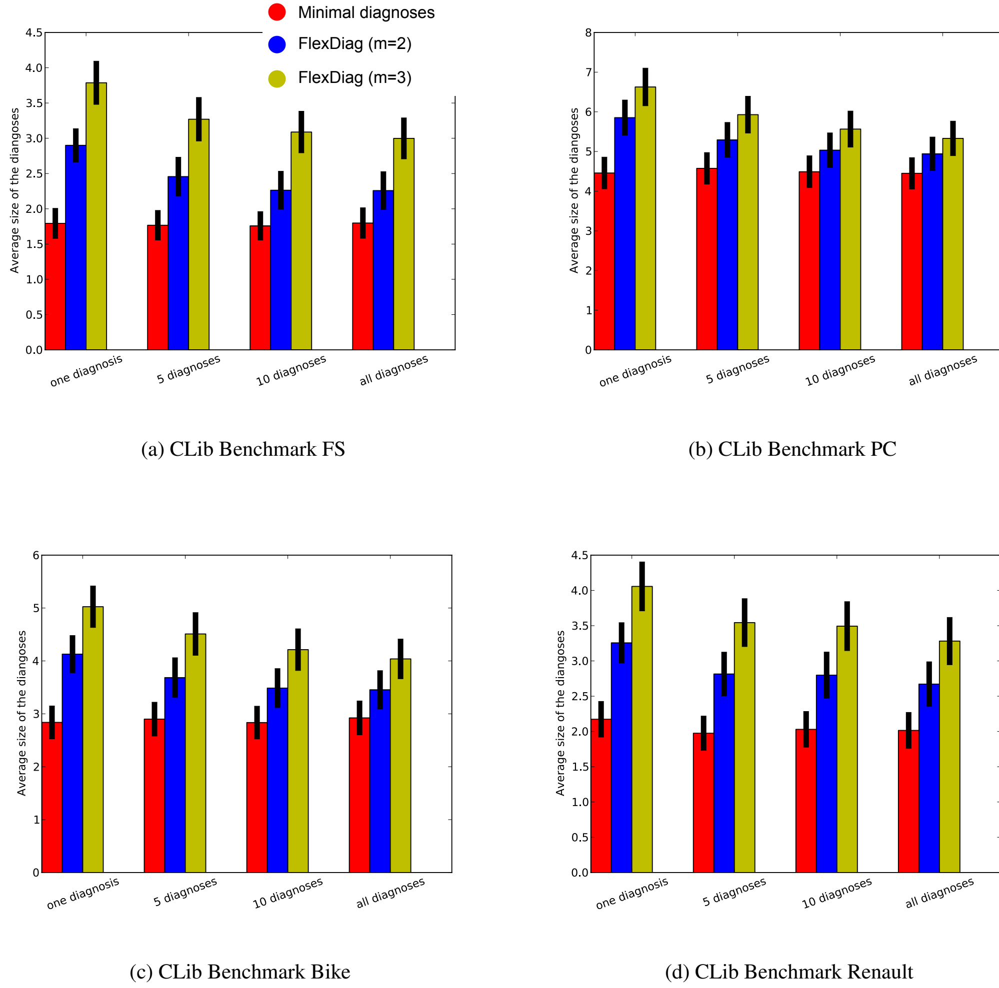
$$MAE(n) = \frac{1}{n}\sum_{i=1}^{n} |1 - position(i)| \tag{4.3}$$

The *position(i)* denotes the position of the diagnosis in the ranking of the algorithm that was then selected by the customer. We take the sum over all **n** sessions. The results of the two-sample T-test show that there exists a significant difference between *breadth-first* search and *all other diagnosis methods* (**p<0.05**). The other algorithms have been tested as well, but we did not detect any significant differences in terms of prediction quality.

Relevance of Clusters

The *FlexDiag* algorithm is the only one introduced that does not calculate minimal diagnoses, but diagnoses clusters. Such a cluster contains at least one minimal diagnosis. Nevertheless, it is probable that it also includes other constraints that are not part of the minimal diagnosis. *FlexDiag* leads to a set of constraints C_D where $C_D = D \cup C_{offset}$ where D is the minimal diagnosis and C_{offset} is a set of constraints which are not part of the minimal diagnosis.

This section aims to analyse the quality of *FlexDiag (m=2)* and *FlexDiag (m=3)* in terms of relevance compared to the minimal diagnosis approaches. In order to get an impression of the quality of the *FlexDiag*, the average size of the diagnoses has been calculated for the different settings of the CLib Benchmark introduced in Section 4.5.2. From the results shown in Figure 4.13, it can be seen that the average number of constraints in a diagnosis is between **1.5** and **4.5**. Taking a look at the results of the *CLib Benchmark FS* (see Figure 4.13(a)) setting it can be seen that the average size of diagnoses calculated by the *FlexDiag (m=2)* and *FlexDiag (m=3)* is decreasing when the number of diagnoses is increased. For calculating one diagnosis, the average size of a diagnosis calculated by *FlexDiag (m=2)* is about **2.8** and calculated by *FlexDiag (m=3)* it is about **3.8**. The average size of the minimal diagnosis for these settings is about **1.8**.



(a) CLib Benchmark FS

(b) CLib Benchmark PC

(c) CLib Benchmark Bike

(d) CLib Benchmark Renault

Figure 4.13.: Average size of the diagnoses in different CLib Benchmarks calculated by approaches calculating minimal diagnoses (first bar) as well as by *FlexDiag (m=2)* (second bar) and *FlexDiag (m=3)* (third bar)

This means, that the offset of the the algorithm *FlexDiag (m=2)* consists one constraint on average, whereas offset the algorithm *FlexDiag (m=3)* contains of two constraints on average.

Taking a look at a dataset with larger diagnoses (see, for example, the *PC* dataset in Figure 4.13(b)) it can be seen that the average number of elements in a diagnosis is again higher for the *FlexDiag (m=2)* and *FlexDiag (m=3)* algorithms. The average size of a minimal diagnosis for this setting is **4.5**. For calculating one diagnosis the average size of a diagnosis calculated by the *FlexDiag (m=2)* is **5.8** and for the *FlexDiag (m=3)* it is **6.6**. Thus the offset of the *FlexDiag (m=2)* is about **1.3** and of the *FlexDiag (m=3)* it is **2.3**. Based on this data, the *relevance* can be evaluated. This relevance is defined as:

$$relevance(D, C_D) = \frac{size(D)}{size(C_D)} \tag{4.4}$$

where **size** describes the size of the constraint set, **D** is the minimal diagnosis and $\mathbf{C}_D$ is the diagnosis

calculated by the *FlexDiag*. The relevance for the *FlexDiag (m=2)* algorithm is **64.3%** for the *PC* dataset and **77.6%** for the *Bike* dataset. For the *FlexDiag (m=3)* algorithm the relevance is **47.4%** for the *PC* dataset and **68.2%** for the *Bike* dataset. Generally speaking, the relevance is higher if there are more elements in the diagnoses. For example, if the set of minimal diagnoses incorporates 4 elements on average, then the diagnoses cluster incorporates between 5 (m=2) and 6 (m=3) elements on average. This results in a relevance of about 72%. If the minimal diagnosis incorporates 10 elements and the diagnoses clusters 11.5 on average, then the relevance of this setting is about 87%. The relevance of the second setting (diagnosis with 10 elements) is about 15% higher compared to the first example with 5 elements. The results for the datasets *Bike* (see Figure 4.13(c)) and *Renault* (see Figure 4.13(d)) of the *CLib Benchmark Library* are similar to the *PC* and *Bike* datasets.

4.6. Related Work

In this chapter different approaches that focus on the identification of (minimal) diagnoses have been introduced. In contrast to calculate diagnoses directly, existing approaches typically rely on the existence of (minimal) conflict sets (Felfernig et al., 2010c).

(O'Sullivan et al., 2007) introduce an approach to identify minimal exclusion sets which correspond to the concept of minimal diagnoses (Reiter, 1987). Based on these minimal exclusion sets, the authors of (O'Sullivan et al., 2007) identify *representative explanations*. The representativeness ensures that the set of explanations presented to the customer does not get too large and still has a lot of variety in it. This is achieved by the fact that each constraint that causes a conflict is shown in at least one explanation to the customer. For example, if the following explanations are retrieved: $e_1 = \{c_1, c_2\}$, $e_2 = \{c_1, c_3\}$ and $e_3 = \{c_2, c_3, c_4\}$, then the explanations e_1 and e_3 constitute a set of representative explanations since each constraint (c_1, c_2, c_3, c_4) appears in at least one of the two explanations. The worst-case size of representative explanations is linear in the number of customer requirements (O'Sullivan et al., 2007).

In the field of model-based diagnosis there exist a couple of algorithms (for example, (Greiner et al., 1989; Wotawa, 2001)) that further developed the original algorithm introduced by (Reiter, 1987) for identifying a diagnosis. These approaches focus on the construction of the Hitting Set Directed Acyclic Graph (Reiter, 1987). (Greiner et al., 1989) introduced a correction to the original algorithm to identify all minimal diagnoses. This correction addresses the lack of the original algorithm (Reiter, 1987) which misses some diagnoses under certain conditions. (Fijany and Vatan, 2004) presents an approach to determine all diagnoses on the basis of an integer programming problem. Consequently, the problem is mapped onto a boolean satisfiability and a 0/1 integer problem. An overview of the approaches to improve the identification of diagnoses can be found in (Lin and Jiang, 2003).

(Feldman et al., 2008) introduced a stochastic fault diagnosis approach that uses a greedy stochastic search. This work is similar to the one introduced in (Lin and Jiang, 2002) which determines hitting sets based on genetic algorithms. These approaches improve the performance significantly; nevertheless, there is neither guarantee of identifying only minimal diagnoses, nor a guarantee of completeness. Anyhow, there are approaches to improve the performance through exploiting the structural properties of the problem. (Jannach and Liegl, 2006) exploit the knowledge of database tables in order to identify minimal diagnoses. (Siddiqi and Huang, 2007) take a look at the system models that can be compiled into a tractable

representation (such as DNNF) for the determination of diagnoses. The key idea behind the algorithm introduced by (Siddiqi and Huang, 2007), is to use the abstraction of circuits and diagnoses pretending that only gates in the set of circuits could be faulty. This abstraction reduces the number of variables in the system.

This chapter focused on the calculation of personalized diagnoses. (de Kleer, 1990) introduced a general approach to use probability values to calculate leading diagnoses. A characterization of preferences is given in (Froehlich et al., 1994). The characterization in this work is expressed using preference relations on single diagnoses and logical formulas on groups of diagnoses. Nevertheless, the work of (Froehlich et al., 1994) misses an algorithm to efficiently calculate preferred diagnoses.

4.7. Discussion

This chapter introduced different algorithms for consistency management in configuration systems. Configuration systems assist customers in configuring potentially complex products or services. During the preference elicitation phase customers are repeatedly defining and revising their requirements. During this refinement, situations may occur where no solution exists that completely fulfills the set of requirements (Pu and Chen, 2008). There exist several possibilities to aid customers in inconsistent situations (see Section 2.3). These situations have to be analysed, in order to choose the right algorithm.

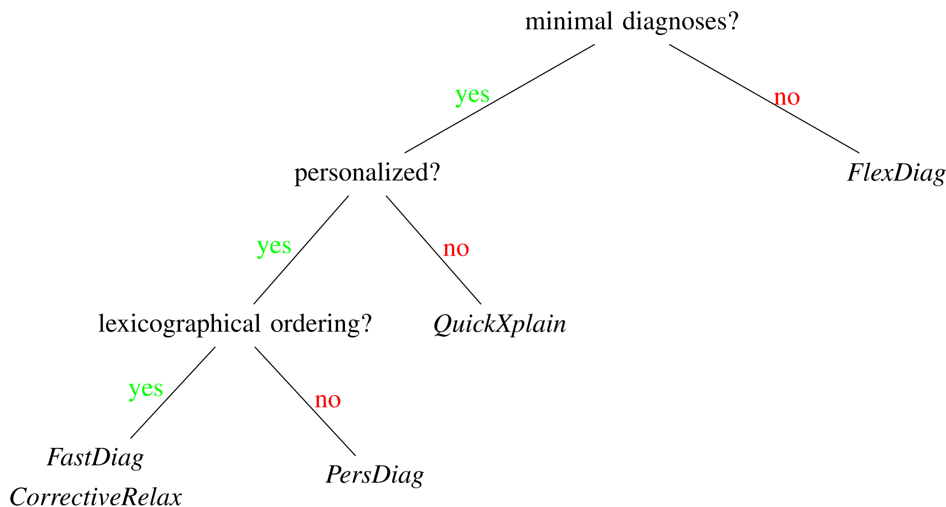


Figure 4.14.: Decision tree to decide what algorithm is suited for the consistency management in configuration systems

In Figure 4.14 a decision tree is presented. This decision tree helps to choose the most suitable algorithm for consistency management when using a configuration system. The first question in the decision tree is *"Should the algorithm calculate minimal diagnoses?"*. If the answer to this question is negative, the *FlexDiag* algorithm is recommended. This recommendation is based on the fact, that the *FlexDiag* algorithm calculates clusters of diagnoses (i.e. non minimal diagnoses). Otherwise (question is answered positively),

another question is asked: *"Should the algorithm be personalized?"*. There exist several algorithms (for example, *FastDiag*, *PersDiag*, *CorrectiveRelax* (O'Callaghan et al., 2005)) that focus on different personalisation strategies. If there is no need for a personalized approach, but it is more important to determine diagnoses ranked by their cardinality, then a combination of *QuickXplain* (Junker, 2004) with the hitting set directed acyclic graph (*HSDAG*) (Reiter, 1987)) is the best choice. This combination of algorithms uses a *breadth-first* approach and is referred as *QuickXplain* in Figure 4.14.

If one is interested in a personalized algorithm to calculate minimal diagnoses then another question is asked: *"Should a lexicographical ordering be used?"*. In the case that there is no lexicographical ordering available, the algorithm *PersDiag* can be suggested, because different personalisation strategies can be used in this algorithm. Otherwise, it can be suggested to use the algorithm *FastDiag* or *CorrectiveRelax* (O'Callaghan et al., 2005) that can take advantage of the lexicographical ordering.

Chapter 5

D-fame: The Diagnosis Framework

This chapter gives an overview of *D-fame: Diagnosis FrAMEwork*. All algorithms introduced so far have been implemented in *D-fame*. This framework has been developed with the goal to compare the algorithms presented in this thesis with the existing state-of-the-art approaches (*QuickXplain* (Junker, 2004), CorrectiveRelax (O'Callaghan et al., 2005), MinRelax (Jannach, 2008)) in terms of performance and prediction accuracy. The *D-fame* framework may serve as a library that offers a set of algorithms that can be applied for consistency management in interactive constraint based systems. The aim was to create an object oriented framework that is easy to extend. The framework is implemented in Java 1.6*.

This chapter is organised as follows: In Section 5.1 an overview of the high-level architecture of the framework is provided. In Section 5.2 more details are provided in order to give the reader an impression of how to use the framework. A summary in Section 5.3 concludes this chapter.

5.1. System Architecture

This section shows the high-level architecture and the main functionality of the *D-fame* framework. The basic architecture consists of four components: (i) the *Input*-component, (ii) the *Algorithm*-component, (iii) the *Datastructure*-component and the (iv) the *Output*-component. The structure of the framework is shown in Figure 5.1. An overview of the different components is given in the following paragraphs.

5.1.1. Input-Component

The *Input*-component is responsible for managing the input for the different conflict detection and diagnosis algorithms. An important input is, for example, the set of constraints which represent customer requirements. Another one is the reasoning engine, which is used for the consistency checks. And last, but not least for some algorithms different personalization strategies can be applied. Therefore, one of the implemented personalization strategies (utility-based, similarity-based, probability-based or hybrid) can be selected. The different packages are described in the following paragraphs:

*http://java.sun.com/

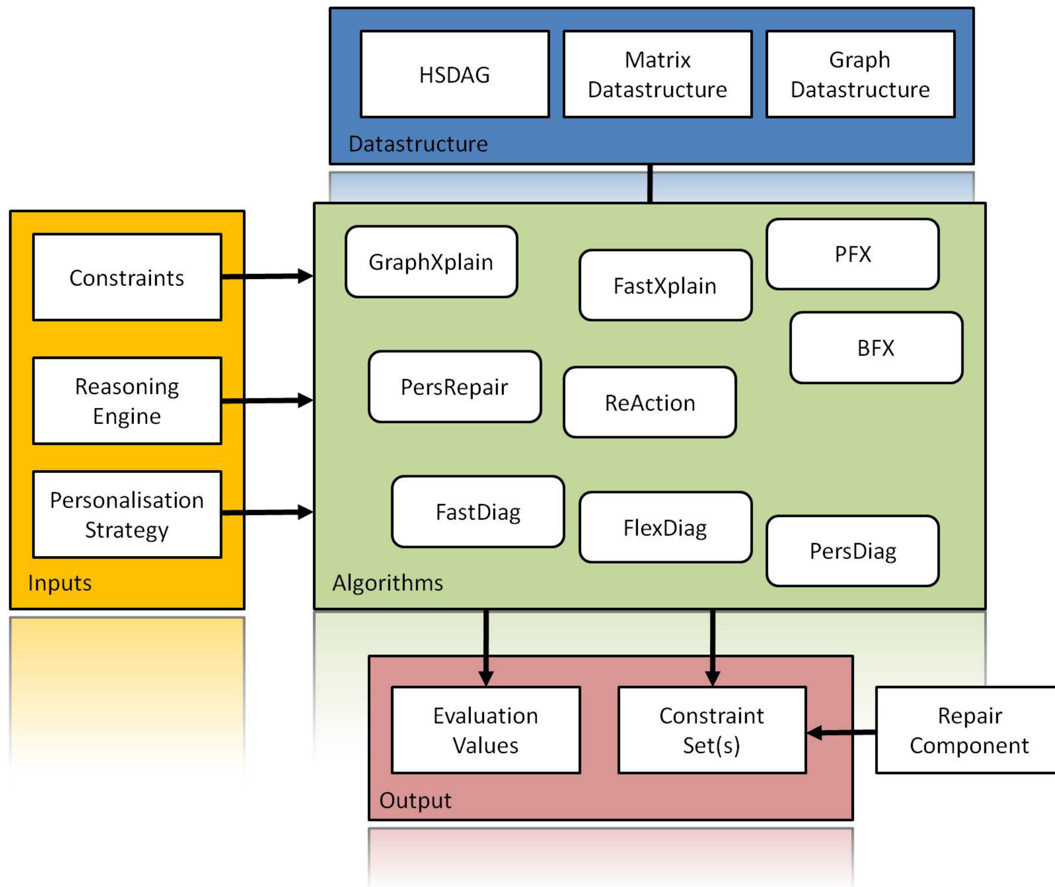


Figure 5.1.: Architecture of the *D-fame* framework. The framework consists of the *Input* component, the *Datastructure* component, the *Output* component and the *Algorithms* component.

Constraints

In order to represent the constraints, we used the abstract class *AbstractConstraint*. This class holds the basic information about a constraint, namely the *name*, a *weight* and the actual *constraint* as an object. From the class *AbstractConstraint*, the classes *HsdagConstraint* and *MdsConstraint* have been derived. The *HsdagConstraint* can be used for every algorithm that can be combined with the hitting set directed acyclic graph (HSDAG) (Reiter, 1987). The *MdsConstraint* is used for the algorithms that operate on the *MatrixDatastructure* (also referred as Intermediate Representation see Section 3.1.2). This type of constraints (*MdsConstraint*) is also used for *GraphXplain*.

Reasoning Engines

Reasoning engines are used for consistency checking. Each reasoning engine has to implement the abstract class *AbstractReasoningEngine*. This *AbstractReasoningEngine* incorporates the method *isCoherent* that takes a collection of constraints and returns *true*, if the set of constraints is consistent, otherwise it returns *false*. In addition, the reasoning engine stores the number of calls (= consistency checks) in a corresponding

log file. This information was used for the performance evaluation of the algorithms introduced in Chapter 4.

For the consistency check different technologies can be used. The *D-fame* framework currently includes the following reasoning components:

- the constraint solvers Choco[†] and Jacop[‡] for constraint satisfaction problems
- for recommendation problems database queries on the MySQL[§] and H2[¶]
- for configuration problems reasoning engines for the CLab (Jensen, 2004) and *SModels* (Tiihonen et al., 2003) systems
- the *MatrixDatastructure* (also called intermediate data structure) can also be used for checking the consistency

Depending on the problem and the available data structures different reasoning engines can be applied. For recommendation problems SQL engines or the matrix data structure are used. Configuration problems are often solved on the basis of constraint satisfaction algorithms. The *D-fame* framework also includes a configurator for the CLab models as well as a configurator for *SModels* which are used for the evaluation with the *WeCoTin* models (Tiihonen et al., 2003). The *SModels* system is an answer set programming (ASP) (Balduccini et al., 2006; Gelfond, 2007) implementation. If a new technology for the consistency checks should be used, it is just needed to override the *AbstractReasoningEngine*.

Personalisation Strategy

Different strategies for personalizing the diagnosis selection process have be introduced in Section 4.4.1. These strategies (utility, probability, and similarity) are operating on the basis of weights. Different ways to calculate the weight of a path are offered. One of the following alternatives can be used (see also Figure 5.2):

- *SingleConstraintWeight*: the weight of the path is the same as the weight of the constraint which is the last element of the path
- *PathWeight*: the weight is the sum of all elements (constraints) of the path
- *NormalizedPathWeight*: this weight is similar to the *PathWeight*, but it is normalized to the length of the path. Thus the *PathWeight* is divided by the number of elements of the path.
- *DenominatorPathWeight*: this weight is the sum of the inverse weight of each constraint of the path ($\sum \frac{1}{w}$). This calculation can be used for weights, where the higher weights lead to less relevant alternatives.

All these weights implement the abstract class *AbstractWeight* which provides an abstract method to calculate the weight for a set of constraints. Additionally to the specification of how the weight of a path is calculated, two ways of sorting the weights are offered. The weights can be either sorted according to

[†] http://www.emn.fr/z-info/choco-solver/
[‡] http://jacop.osolpro.com/
[§] http://www.mysql.com/
[¶] http://www.h2database.com

highest weight first or lowest weight first. If, for example, the *PathWeight* type of calculation is applied, the algorithm prefers paths with a low utility, because these include the constraints a customer is more likely to adapt. On the other hand, if the algorithm is interested in requirements with a high probability of being changed by the customer, paths that include the corresponding requirements should be returned first.

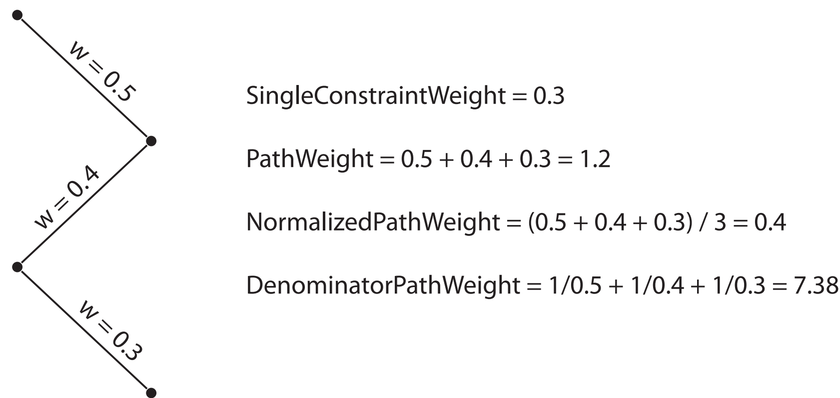


Figure 5.2.: Overview of different weight calculations (*SingleConstraintWeight*, *PathWeight*, *Normalized-PathWeight* and *DenominatorPathWeight*) based on a path with three constraint weights (0.5, 0.4, and 0.3).

5.1.2. Algorithms-Component

The following algorithms are provided by *D-fame* to aid customers in the consistency management when interacting with recommender systems:

- *GraphXplain*, an algorithm with the aim of calculating minimal conflict sets using a one-mode network (see Section 3.2)
- *FastXplain*, an algorithm with the aim of calculating minimal conflict sets. At the same time it identifies diagnoses (see Section 3.3)
- *BFX*, an improved version of the *FastXplain* (see Section 3.4)
- *PFX*, a personalized version of the *BFX* (see Section 3.5)
- *PersRepair*, an algorithm with the aim of calculating minimal diagnoses in a personalized fashion (see Section 3.6)
- *ReAction*, an algorithm with the aim to calculate repair actions based on a sorted set of diagnoses (see Section 3.7)

In addition, *D-fame* contains a set of algorithms to support customers in the consistency management when interacting with configuration systems:

- *FastDiag*, an algorithm with the aim of calculating minimal diagnoses in a personalized fashion (see Section 4.2)

- *FlexDiag*, an algorithm to identify clusters which include at least one diagnosis (see Section 4.3)
- *PersDiag*, an algorithm with the aim to calculate diagnoses in a personalized fashion (see Section 4.4)

In addition to the listed approaches, the *D-fame* framework offers a set of further state-of-the-art diagnosis and conflict detection algorithms.

- *MinRelax*, an algorithm to calculate minimal relaxations similar to diagnoses (Jannach, 2008)
- *CorrectiveRelax*, an algorithm to calculate corrective explanations similar to diagnoses (O'Callaghan et al., 2005)
- *QuickXplain*, an algorithm to identify minimal conflict sets (Junker, 2004)

Furthermore, we implemented the hitting set directed acyclic graph (HSDAG) (Reiter, 1987) which can be combined with the mentioned algorithms.

5.1.3. Datastructure-Component

The algorithms included in *D-fame* operate on different data structures. *FastXplain*, *BFX*, *PFX* and *MinRelax* use the intermediate data structure (see Sections 3.3-3.5), whereas other algorithms can be combined with the hitting set directed acyclic directed graph (HSDAG) (Reiter, 1987). *GraphXplain* builds up a graph from the package *Graph Datastructure*. Major elements of the data structure component are discussed in more detail in the following.

HSDAG

The hitting set directed acyclic graph (HSDAG) (Reiter, 1987) is an approach to derive minimal diagnoses based a on minimal conflict sets (see also Section 3.6). With the same approach, it is possible to derive minimal conflict sets based on minimal diagnoses (see Section 3.3). Note that we also implemented the pruning functionality (Greiner et al., 1989; Wotawa, 2001). In the *D-fame* framework every algorithm that can identify one conflict set or one diagnosis, can be used in combination with the HSDAG. *ReAction*, *PersRepair*, *FastDiag*, *FlexDiag*, *PersDiag*, *QuickXplain* and *CorrectiveRelax* are combined with the HSDAG in the framework. If an algorithm uses the HSDAG, the constraints must be of the type *HsdagConstraint*.

Intermediate Data Structure

The intermediate data structure (also called matrix data structure) as described in Section 3.1.2 represents consistency relationships between customer requirements and products. This data structure can only be applied in the context of recommendation problems, where the set of possible products is defined in an explicit fashion.

The intermediate data structure can also be used for consistency checking (on the basis of the *MatrixDatastructure* reasoning engine). In order to check the consistency, the *MatrixDatastructure* reasoning engine needs to check whether there exists at least one product that satisfies all the given requirements.

Graph Data Structure

The graph data structure builds a graph based on the information of the intermediate data structure. The intermediate data structure can be interpreted as a two-mode network. This two-mode network can be folded into a one-mode network (for a further description see Section 3.2). The one-mode network is used by the *GraphXplain* algorithm to identify minimal conflict sets.

5.1.4. Output-Component

The main output of the different algorithms are constraint set(s). These constraint set(s) can be either conflict sets or diagnoses. In addition to these constraints sets, meta information is provided (e.g., algorithm run time and number of solver calls) which can be exploited for evaluation purposes.

Constraint Set(s)

Each algorithm provides a method to retrieve the calculated sets of constraints. These sets can either be minimal conflict sets or minimal diagnoses. For identifying repair actions, the *RepairComponent* needs to be activated. This component requires a diagnosis and knowledge about the product assortment as inputs. The output of this component is a set of repair assignments to all attributes.

Evaluation Values

The evaluation values can be used for performance evaluation. The most important ones are the run time and number of theorem prover calls. Additionally to the determination of the evaluation values, the *D-fame* framework provides the possibility to store these values either in a file or in a database. This is very practical for later evaluations and plots.

5.2. Using the framework

With the goal to provide insights in how to apply the D-fame framework, this section presents some key code snippets. Generally speaking, the framework is easy to use and implemented in an object-oriented fashion, such that it can be easily extended.

Constraints

As the constraints are the central objects, we first show how to create them. An important class is the abstract class, *AbstractConstraint*, from which the classes *MdsConstraint* and *HsdagConstraint* are derived. Both constraint types require three parameters, namely the *name*, a *weight* and an *object* that represents the constraint in a form applicable for the reasoning engine. The *MdsConstraint* can be created as follows:

```
AbstractConstraint c_mds = new MdsConstraint(name, weight, object);
```

A set of *MdsConstraint* must be created when the matrix data structure (intermediate data structure) is used. When the HSDAG is used, the constraints are of type *HsdagConstraint*. *HsdagConstraints* can be created as follows:

```
AbstractConstraint c_hsdag = new HsdagConstraint(name, weight, object);
```

Reasoning Engines

Various reasoning engines are provided by *D-fame*. Most of these reasoning engines do not require parameters. An exception is the *MatrixReasoningEngine*, which creates an intermediate data structure and performs the consistency checks on this. The *MatrixReasoningEngine* can be created as follows:

```
engine = MatrixReasoningEngine(requirements, products)
```

The *SQLReasoningEngine* requires a connection to the database as parameter. This connection must be configured with all values that are needed to connect to the database. In addition, it needs to be configured with the name of the database as well as with the user name and the password in order to actually connect to the database. The *SQLReasoningEngine* can be created as follows:

```
engine = new SQLReasoningEngine(connection);
```

Algorithms

All algorithms implemented in the *D-fame* framework are derived from the class *AbstractAlgorithm*. The *FastXplain* algorithm can be instantiated as follows:

```
AbstractAlgorithm algorithm = new FastXplain(requirements, products);
```

FastXplain requires two parameters: the requirements of the customer and a list of products (stored in the class *Products*). Based on this information the algorithm can be activated as follows:

```
algorithm.run();
```

In a default configuration, the algorithm calculates all minimal conflict sets and all minimal diagnoses. If the number of minimal conflict sets or minimal diagnoses should be restricted, this can be achieved as follows:

```
algorithm.configure(target_diagnoses, target_mcs);
```

This can be applied to any algorithm. Nevertheless, it needs to be done before the algorithm is activated.

The algorithm *GraphXplain* (see Section 3.2) is initialized in a similar way as the algorithm *FastXplain*.

```
AbstractAlgorithm algorithm = new GraphXplain(requirements, products);
```

Compared to this, the *FastDiag* requires only one parameter - the reasoning engine. Additionally to the instantiation of the algorithm, *FastDiag* needs to be configured with the parameters **c** (all constraints including the configuration knowledge base and customer requirements) and **c_r** (customer requirements). Based on these parameters the diagnoses are determined.

The instantiation of the *FastDiag* is done as follows:

```
AbstractAlgorithm algorithm = new FastDiag(engine);

algorithm.configure(c, c_r);
```

Similar to the instantiation of the *FastDiag* algorithm, the *PersDiag* algorithm can be created. *PersDiag* requires three parameters: the reasoning *engine*, the personalization *criteria* and the *sorting*. With the criteria it can be specified which personalization strategy (for example, utility, probability) should be used. The *sorting* parameter indicates if the path with the highest weight first or with the lowest weight should be considered. Additionally to the instantiation of the algorithm, *PersDiag* needs to be configured with the parameters **c_kb** (constraints of configuration knowledge base) and **c_r** (customer requirements). Based on these parameters the diagnoses are determined.

The instantiation of the *PersDiag* is done as follows:

```
AbstractAlgorithm algorithm = new PersDiag(engine, criteria, sorting);

algorithm.configure(c_kb, c_r);
```

5.3. Summary

This chapter introduced the *D-fame: Diagnoses FrAMEwork*. The framework was built to compare different diagnoses detection algorithms and can be used either as a framework for research purposes or as a library that provides algorithms for the consistency management in constraint-based systems. The framework has been developed in an object-oriented way and it is easy to extend. In this chapter, an overview (*D-fame* framework architecture) has been provided.

Chapter 6

Conclusion and Future Work

For a constraint-based system that aims to satisfy all constraints or requirements of the customer, it is important to include technologies and algorithms for the consistency management. This thesis introduces and evaluates approaches that can be applied in constraint-based recommender systems and in knowledge-based configuration systems. The final chapter reflects on important research questions. Furthermore, a decision tree is included that helps to identify the algorithm that is suited best for a specific problem. Finally, this chapter highlights further research directions.

6.1. Conclusion

This thesis has addressed the following research questions (for a more detailed explanation see Section 1.2). In the following paragraphs, it is outlined how these questions have been answered:

Research Question Q1:

How can structural properties of knowledge-based recommender systems be used to improve the run time performance of conflict detection algorithms?
For addressing this research question a couple of algorithms that explore the structural properties of knowledge-based recommender systems have been introduced. First, the *GraphXplain* (see Section 3.2) algorithm was presented. This approach creates a one-mode network of customer requirements in order to determine minimal conflict sets from this. Another approach is the *FastXplain* algorithm which uses the intermediate data structure (Jannach, 2008) constructed from the product assortment and the customer requirements to derive minimal diagnoses. Using the concepts introduced by (Reiter, 1987) the *FastXplain* algorithm derives minimal conflict sets. This approach was further developed which resulted in *BFX* (Boosted FastXplain). In another extension of the *FastXplain* algorithm a personalization strategy was added (*PFX* (Personalized FastXplain)).

Research Question Q2:

How can customers be supported in restoring consistency between the requirements and the corresponding product assortment?
Customers can be aided by an iterative presentation of minimal conflict sets (explicitly calculated by the algorithms *GraphXplain*, *FastXplain*, *BFX* and *PFX)*. Another approach is to present a list of diagnoses to the customer (explicitly calculated by, for example, *PersDiag*). Furthermore, customers can be aided in an inconsistent situation with a list of repair actions (explicitly calculated by the algorithms *PersRepair* and *ReAction*). A repair action is an action that can be performed by the customer in order to restore consistency. For more details see Section 2.3.

Research Question Q3:

What are the possible run time improvements for diagnosis algorithms to support customers, who are interacting with a constraint-based system?
There are several possibilities to improve the current state-of-the-art algorithms in terms of run time performance. All algorithms presented in this work perform better compared to existing approaches at least in some situations. The algorithms *ReAction* (see Section 3.7) and *FastDiag* (see Section 4.2) are especially interesting, due to their general applicability. Another approach is the *FlexDiag* (see Section 4.3) which is a fast approach for calculating a small number of diagnoses. Nevertheless, it is not guaranteed that the algorithm finds minimal diagnoses. A detailed evaluation of the improvements is presented in Section 3.8.1 and Section 4.5.2.

Research Question Q4:

How can diagnoses be personalized in order to achieve a high prediction accuracy?
This work focuses on the application of different personalization strategies. The algorithm *PFX* uses utility values, whereas the algorithm *PersRepair* (see Section 3.6) uses different similarity measures. Moreover, the algorithm *PersDiag* (see Section 4.4) uses different strategies such as a utility-based, a similarity-based, a probability-based, and a hybrid strategy. A lexicographical ordering based on utility values is used by the algorithms *ReAction* (see Section 3.7) and *FastDiag* (see Section 4.2). A detailed evaluation of the improvements is presented in Sections 3.8.2 and 4.5.3.

6.2. Overview of the Algorithms

This thesis introduces different technologies and approaches for the consistency management in constraint-based systems. In order to get a better overview, Figure 6.1 shows a decision tree to help to decide which algorithm to apply in which situation. The first question that is asked in this decision tree is *"Should the algorithm be personalized?"*. The *D-fame* framework integrates several algorithms that focus on different personalisation strategies. If the algorithm should have a personalization strategy included, another question is asked, namely *"Is a product table available?"*. Depending on the answer to this question, the algorithm *PFX* (see Section 3.5) is suggested (if a table representation of the items is available). In the case

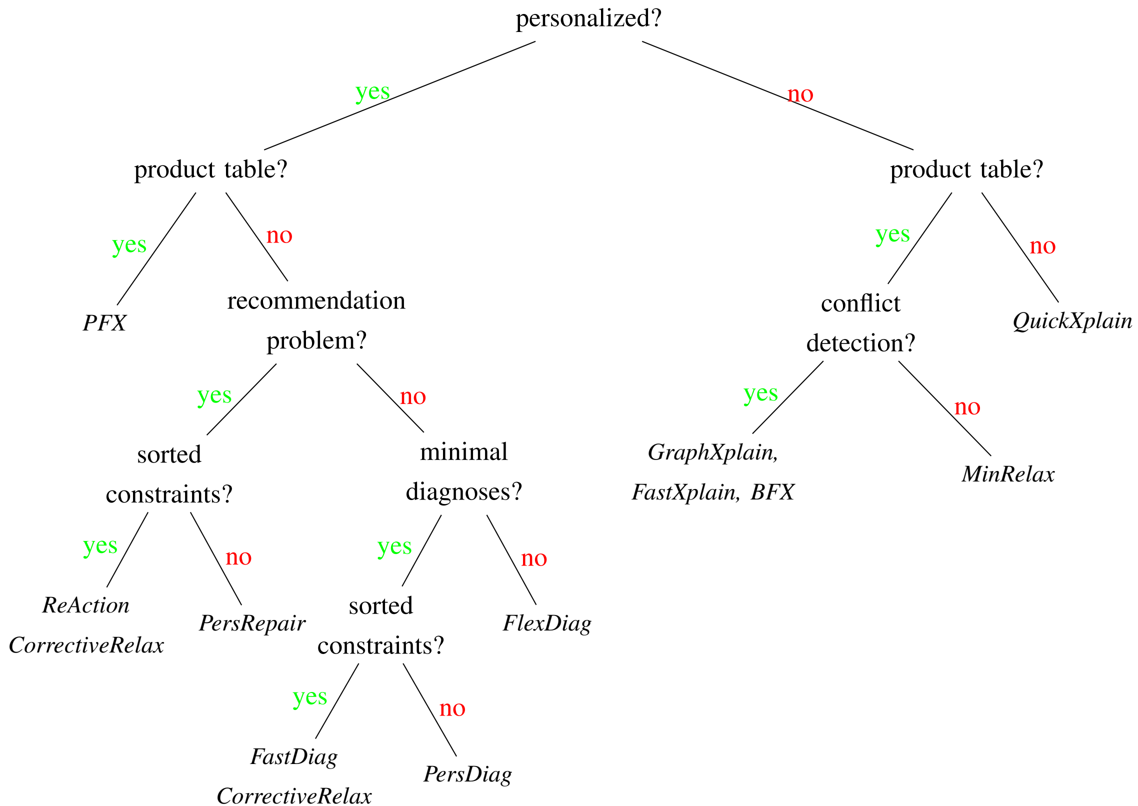


Figure 6.1.: Decision tree to decide what algorithm is suited for the consistency management

that there is no table representation available, other introduced algorithms can be applied. Note that these algorithms can also be applied on a product table, but they are also suited for other representations.

Algorithms for applications that should include a personalization strategy can be derived from the left branch of Figure 6.1). If there is no a product table available, the following question is asked: *"Is the problem a recommendation problem or a configuration problem?"*. If the application is a recommender application, the following algorithms are suggested: *PersRepair* (see Section 3.6), *ReAction* (see Section 3.7), and *CorrectiveRelax* (O'Callaghan et al., 2005). If a set of sorted constraints is available and a few diagnoses should be calculated, the algorithms *ReAction* and *CorrectiveRelax* are recommended to be use. Otherwise, it is suggested to use the algorithm *PersRepair*, where different personalization strategies can be used. If the problem is a configuration problem, the following question is asked *"Is an approach wanted that calculates minimal diagnoses/repair actions?"*. If there is no need for minimal diagnoses, the algorithm *FlexDiag* (see Section 4.3) is suggested, especially when a small number of diagnoses should be determined. In the case that minimal diagnoses are needed, it is suggested to use the *FastDiag* (see Section 4.2) or *CorrectiveRelax* (O'Callaghan et al., 2005) if the constraints are available in a sorted order. Otherwise, using the algorithm *PersDiag* (see Section 4.4) in which different personalization strategies can be included, is feasible.

The algorithms for applications, with no need or no possibility for a personalization approach, can be derived from the right branch of the decision tree shown in Figure 6.1. The question asked, is *"Is a product*

table available?". If there is no product table available, but a model of the products, the *QuickXplain* algorithm can be recommended. This algorithm is a combination of the *QuickXplain* (Junker, 2004) and the Hitting Set Directed Acyclic Graph (*HSDAG*) (Reiter, 1987). This combination performs a breadth-first search in the *HSDAG*. Note that the *QuickXplain* can also be applied in settings where a product table is available. If there is a product table available, other algorithms can also be applied. In order to find a feasible algorithm of the remaining settings, the following question is asked: *"Should the algorithm focus on the calculation of minimal conflict sets or on minimal diagnoses?"*. Depending on the answer to this question, the following algorithms can be suggested: *GraphXplain*, *FastXplain* and *BFX* (if the answer is minimal conflict sets) and the algorithm *MinRelax* (Jannach, 2008) otherwise.

6.3. Future Work

This thesis comprises a set of concepts that address the objectives of consistency management in constraint-based systems. Beneath that, some new and challenging questions emerged from the introduced concepts that might stimulate further research.

Further Algorithms

In this thesis and in the *D-fame* framework different approaches for consistency management in constraint-based systems have been successfully introduced and evaluated. These approaches can be further developed in order to be applied on a wider range of problems and to aid customers in a better way. An approach that would be nice to have, is to extend the approaches in a way that they can run on a distributed computing system. This would allow the handling of more complex situations in less time, due to the increased processing power.

Another improvement that lies within future work, is the development of algorithms to cluster constraints in a way that they most probably incorporate a diagnosis. Such approaches can be used to preselect a set of constraints that should be diagnosed. A similar approach would be to select a constraint that should be in the diagnosis, before the algorithm calculates the diagnosis. The goal of this approach is to find one or all diagnoses that contain the selected constraint without calculating other diagnoses. For solving this challenge, the introduced approaches need to be further developed.

Another further development, especially of the intermediate representation (see Section 3.1.2), is to find a method to create (binary) decision diagrams. Extensions to our algorithms can help derive diagnoses from these decision diagrams.

Improving Personalization Strategies

The current version of the *D-fame* Framework integrates an initial version of personalization strategies. In a further extension, an evaluation of additional personalization strategies should be performed. This may include, for example, an extension of the algorithms *PersRepair* and *PersDiag* with an adapted probability-based approach. This approach would calculate the probability values on a reduced set of log entries. This reduction of all log entries may be performed using similarity measures.

Another possible improvement lies in an extensive evaluation of different hybrid approaches. The evaluation may compare the prediction quality of the different hybrid approaches based on the different personalization strategies.

Moreover, learning the weights which are used for the different personalization strategies (especially for the algorithms *ReAction* and *FastDiag*) may also lead to a better prediction quality. An evaluation comparing the original approaches (as described in Section 3.7 and 4.2) and the ones using learned weights should be performed. Furthermore, for learning the weights different learning strategies, for example, reinforcement learning, bayesian networks, or neuronal networks, may be used. The *D-fame* framework provides a sound base for these future evaluations, since the algorithms are already implemented and can easily be extended with additional personalization strategies.

Psychological Aspects

This thesis left out the psychological aspects of influencing customers in constraint-based systems. These psychological aspects cover, for example, the decision bias, decision making, design of user interfaces, cognitive aspects, decoy effects and defaults (see, for example, (Asch, 1949; Payne, 1976; Bettman and Kakkar, 1977; Lussier and Olshavsky, 1979; Tversky and Kahneman, 1981; Samuelson and Zeckhauser, 1988; Mandl et al., 2011)). An evaluation that coveres the acceptence of the introduced approaches with regard to different psychological aspects lies within future work.

Further Applications

The introduced approaches have been evaluated on a wide range of different settings. Nevertheless, an evaluation studying the applicability in the context of an industrial system over a longer period of time is still missing. For this evaluation the *D-fame* framework should be used as all introduced algorithms are already implemented.

The approaches for identifying consistent configurations can also be applied in collaborative systems (i.e. requirements engineering, software development (Felfernig et al., 2010e,d)). In these systems inconsistencies can emerge, when different users have different opinions, but need to select the best option. In such inconsistent situations, the approaches introduced in this thesis can be applied.

Bibliography

ADOMAVICIUS, G. AND TUZHILIN, A. 2005. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering 17,* 6, 734–749.

ANAND, S. S. AND MOBASHER, B. 2005. Intelligent techniques for web personalization. *Lecture Notes in Computer Science 3169*, 1–36.

ARDISSONO, L., FELFERNIG, A., FRIEDRICH, G., GOY, A., JANNACH, D., PETRONE, G., SCHÄFER, R., AND ZANKER, M. 2003. A framework for the development of personalized, distributed web-based configuration systems. *AI Magazine 24,* 3, 93–108.

ASCH, S. 1949. Forming impressions of personality. *Journal of Abnormal and Social Psychology 41*, 258–290.

BAEZA-YATES, R. A. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BAKKER, R. R., DIKKER, F., TEMPELMAN, F., AND WOGMIM, P. M. 1993. Diagnosing and solving over-determined constraint satisfaction problems. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 276–281.

BALABANOVIĆ, M. AND SHOHAM, Y. 1997. Fab: content-based, collaborative recommendation. *Communications of the ACM 40*, 66–72.

BALDUCCINI, M., GELFOND, M., AND NOGUEIRA, M. 2006. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence 47*, 183–219.

BARKER, V. E., O'CONNOR, D. E., BACHANT, J., AND SOLOWAY, E. 1989. Expert systems for configuration at digital: Xcon and beyond. *Communication ACM 32*, 298–318.

BELANGER, F. 2005. A conjoint analysis of online consumer satisfaction. *Journal of Electronic Commerce Research 6*, 95–111.

BELL, R. M. AND KOREN, Y. 2007. Improved neighborhood-based collaborative filtering. In *1st KDD-Cup'07*. San Jose, California.

BETTMAN, J. AND KAKKAR, P. 1977. Effects of information presentation format on consumer information acquisition strategies. *Journal of Consumer Research 3*, 233–240.

BEUCHE, D., PAPAJEWSKI, H., AND SCHRÖDER-PREIKSCHAT, W. 2004. Variability management with feature models. *Science of Computer Programming - Special issue: Software variability 53*, 333–352.

BRIDGE, D., GÖKER, M. H., MCGINTY, L., AND SMYTH, B. 2005. Case-based recommender systems. *Knowledge Engineering Review 20*, 315–320.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers 35*, 677–691.

BURKE, R. 2000. Knowledge-based recommender systems. *Encyclopedia of Library and Information Systems 69,* 32, 180–200.

BURKE, R. 2002a. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction 12*, 331–370.

BURKE, R. 2002b. Interactive critiquing forcatalog navigation in e-commerce. *Artificial Intelligence Review 18*, 245–267.

BURKE, R., FELFERNIG, A., AND GOEKER, M. to appear 2011. *AI Magazine*, The Future of Recommender Systems: Research and Applications ed. AAAI.

BURKE, R. D., HAMMOND, K. J., AND YOUNG, B. C. 1997. The findme approach to assisted browsing. *IEEE Expert: Intelligent Systems and Their Applications 12*, 32–40.

CARD, S. K., ROBERTSON, G. G., AND MACKINLAY, J. D. 1991. The information visualizer, an information workspace. In *Proceedings of the Conference on Human Factors in Computing Systems: Reaching Through Technology*. ACM, New York, NY, USA, 181–186.

CATILLO, L. 2005. *Planning, Scheduling And Constraint Satisfaction: From Theory To Practice*. Ios Pr Inc.

CHAKRABARTI, S. 2002. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kaufmann Publishers.

CLAYPOOL, M., GOKHALE, A., MIRANDA, T., MURNIKOV, P., NETES, D., AND SARTIN, M. 1999. Combining content-based and collaborative filters in an online newspaper. In *In Proceedings of ACM SIGIR Workshop on Recommender Systems*.

CUNIS, R., GÜNTER, A., AND STRECKER, H., Eds. 1991. *Das PLAKON-Buch, Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen*. Springer-Verlag, London, UK.

CUNIS, R., GÜNTER, A., SYSKA, I., PETERS, H., AND BODE, H. 1989. Plakonan approach to domain-independent construction. In *Proceedings of the 2nd International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. ACM, New York, NY, USA, 866–874.

DAVIS, S. M. 1989. From future perfect: Mass customizing. *Strategy and Leadership 17,* 2, 16–21.

DE KLEER, J. 1990. Using crude probability estimates to guide diagnosis. *Artificial Intelligence 45,* 3 (October), 381–391.

DE KLEER, J., MACKWORTH, A. K., AND REITER, R. 1992. *Characterizing diagnoses and systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 54–65.

DE KLEER, J. AND WILLIAMS, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence 32,* 1, 97–130.

FELDMAN, A., PROVAN, G., AND GEMUND, A. V. 2008. Computing minimal diagnoses by greedy stochastic search. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI).* AAAI Press, 911–918.

FELFERNIG, A. 2007. Reducing development and maintenance efforts for web-based recommender applications. *International Journal of Web Engineering Technologies 3*, 329–351.

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. ACM, New York, NY, USA, 1–10.

FELFERNIG, A., FRIEDRICH, G., JANNACH, D., AND STUMPTNER, M. 2004. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence 152,* 2, 213–234.

FELFERNIG, A., ISAK, K., AND KRUGGEL, T. 2005. Testing knowledge-based recommender applications. *ÖGAI-Journal 24,* 4, 1–7.

FELFERNIG, A., MANDL, M., AND SCHUBERT, M. 2010a. Intelligent debugging of utility constraints in configuration knowledge bases. In *Informatik 2010: Service Science - Neue Perspektiven für die Informatik, Beiträge der 40. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Band 1, 27.09.-1.10.2010, Leipzig*. LNI. GI, 767–772.

FELFERNIG, A., MANDL, M., TIIHONEN, J., SCHUBERT, M., AND LEITNER, G. 2010b. Personalized user interfaces for product configuration. In *Proceeding of the 14th International Conference on Intelligent User Interfaces*. IUI '10. ACM, New York, NY, USA, 317–320.

FELFERNIG, A. AND SCHUBERT, M. 2010a. Diagnosing inconsistent requirements. In *Proceedings of the Workshop on Configuration (affiliated with ECAI'10).* Lisbon, Portugal, 15–20.

FELFERNIG, A. AND SCHUBERT, M. 2010b. A diagnosis algorithm for inconsistent constraint sets. In *Proceedings of the 21st International Workshop on the Principles of Diagnosis*. 31–38.

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., SCHUBERT, M., FRIEDRICH, G., MAIRITSCH, M., AND MANDL, M. 2009b. Personalized diagnosis and repair of customer requirements in constraint-based recommendation. In *DX-09 - 20th International Workshop on Principles of Diagnosis*. 315–320.

FELFERNIG, A., SCHUBERT, M., FRIEDRICH, G., MANDL, M., MAIRITSCH, M., AND TEPPAN, E. 2009c. Plausible repairs for inconsistent requirements. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. Pasadena, California, USA, 791–796.

FELFERNIG, A., SCHUBERT, M., MANDL, M., FRIEDRICH, G., AND TEPPAN, E. 2010c. Efficient explanations for inconsistent constraint sets. In *Proceedings of the ECAI 2010 - 19th European Conference on Artificial Intelligence*. Vol. 215. IOS Press, Lisbon, Portugal, 1043–1044.

FELFERNIG, A., SCHUBERT, M., MANDL, M., AND GHIRARDINI, P. 2010d. Diagnosing inconsistent requirements preferences in distributed software projects. In *Software Engineering 2010 - Workshopband.* LNI, vol. 160. GI, 495–502.

FELFERNIG, A., SCHUBERT, M., MANDL, M., RICCI, F., AND MAALEJ, W. 2010e. Recommendation and decision technologies for requirements engineering. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. RSSE '10. ACM, New York, NY, USA, 11–15.

FELFERNIG, A., SCHUBERT, M., AND ZEHENTNER, C. 2011. An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing (AIEDAM) 25,* 2, 175–184.

FIJANY, A. AND VATAN, F. 2004. New approaches for efficient solution of hitting set problem. In *Proceedings of the International Synposium on Information and Communication Technologies*. Trinity College Dublin, 1–10.

FINKELSTEIN, A. 2000. A foolish consistency: Technical challenges in consistency management. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*. Springer-Verlag, London, UK, 1–5.

FLEISCHANDERL, G., FRIEDRICH, G. E., HASELBÖCK, A., SCHREINER, H., AND STUMPTNER, M. 1998. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems 13*, 59–68.

FRAYMAN, F. AND MITTAL, S. 1987. Cossack: A constraint-based expert system for conguration tasks. In *Knowledge-Based Expert Systems in Engineering, Planning and Design*. 143–166.

FREUDER, E. C. 1997. In pursuit of the holy grail. *Constraints: An International Journal 2*, 57–61.

FRIEDRICH, G. AND SHCHEKOTYKHIN, K. M. 2005. A general diagnosis method for ontologies. In *International Semantic Web Conference*. Lecture Notes in Computer Science, vol. 3729. Springer, 232–246.

FROEHLICH, P., NEJDL, W., AND SCHROEDER, M. 1994. A formal semantics for preferences and strategies in model-based diagnosis. In *5th International Workshop on Principles of Diagnosis (DX'94)*. 106–113.

GELFOND, M. 2007. *In Handbook of Knowledge Representation*. Elsevier Science, Chapter Answer Sets.

GLENBERG, A., BRADLEY, M., STEVENSON, J., KRAUS, T., TKACHUK, M., AND GRETZ, A. 1980. A two-process account of long-term serial position effects. *Journal of Experimental Psychology: Human Learning and Memory 6,* 4, 355369.

GODFREY, P. 1997. Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems 6,* 2, 95–149.

GOKNIL, A., KURTEV, I., AND BERG, K. 2008. A metamodeling approach for reasoning about requirements. In *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*. Springer-Verlag, Berlin, Heidelberg, 310–325.

GOLDBERG, D., NICHOLS, D., OKI, B. M., AND TERRY, D. 1992. Using collaborative filtering to weave an information tapestry. *Communications of the ACM 35*, 61–70.

GREINER, R., SMITH, B. A., AND WILKERSON, R. W. 1989. A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence 41*, 79–88.

HAAG, A. 1998. Sales configuration in business processes. *IEEE Intelligent Systems 13*, 78–85.

HAAG, A. 2005. Dealing with configurable product in the sap business suite. In *Papers from the Configuration Workshop at IJCAI'05*. 68–71.

HADZIC, T., SUBBARAYAN, S., JENSEN, R. M., ANDERSEN, H. R., MØLLER, J., AND HULGAARD, H. 2004. Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of the International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems*. 131–138.

HALES, H. 1992. Automating and integrating the sales function: How to profit from complexity and customization. *Enterprise Integration Strategies 9,* 11, 1–9.

HAMMOND, K. J., BURKE, R., AND LYTINEN, S. L. 1995. A case-based approach to knowledge navigation. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2071–2072.

HAMSCHER, W., CONSOLE, L., AND DE KLEER, J., Eds. 1992. *Readings in model-based diagnosis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

HERLOCKER, J. L., KONSTAN, J. A., TERVEEN, L. G., AND RIEDL, J. T. 2004. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS) 22*, 5–53.

HILL, W., STEAD, L., ROSENSTEIN, M., AND FURNAS, G. 1995. Recommending and evaluating choices in a virtual community of use. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 194–201.

HUFFMAN, C. AND KAHN, B. E. 1998. Variety for sale: Mass customization or mass confusion? *Journal of Retailing 74,* 4, 491 – 513.

JANNACH, D. 2008. Finding preferred query relaxations in content-based recommenders. In *Intelligent Techniques and Tools for Novel System Architectures*. Studies in Computational Intelligence, vol. 109. Springer, Berlin / Heidelberg, 81–97.

JANNACH, D. AND LIEGL, J. 2006. Conflict-directed relaxation of constraints in content-based recommender systems. In *Advances in Applied Artificial Intelligence*. Lecture Notes in Computer Science, vol. 4031. Springer Berlin / Heidelberg, Berlin / Heidelberg, 819–829.

JANNACH, D., ZANKER, M., FELFERNIG, A., AND FRIEDRICH, G. 2011. *Recommender Systems An Introduction*. Cambridge University Press.

JENSEN, R. 2004. *CLab user manual*, ITU-TR-2004-46 ed. IT University of Copenhagen.

JOACHIMS, T., GRANKA, L., PAN, B., HEMBROOKE, H., AND GAY, G. 2005. Accurately interpreting clickthrough data as implicit feedback. In *Proceedings of the 28th Annual International Conference on Research and Development in Information Retrieval*. ACM, New York, NY, USA, 154–161.

JUNKER, U. 2004. Quickxplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artifical intelligence*. AAAI'04. AAAI Press, 167–172.

JUNKER, U. AND MAILHARRO, D. 2003. The logic of ilog (j)configurator: Combining constraint programming with a description logic. In *Proceedings of the IJCAI-03 Configuration Workshop*. 13–20.

KONSTAN, J. A., MILLER, B. N., MALTZ, D., HERLOCKER, J. L., GORDON, L. R., AND RIEDL, J. 1997. Grouplens: applying collaborative filtering to usenet news. *Communications of the ACM 40*, 77–87.

LIFFITON, M. H. AND SAKALLAH, K. A. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning 40*, 1–33.

LIN, L. AND JIANG, Y. 2002. Computing minimal hitting sets with genetic algorithm. *Algorithmica 31,* 2, 95–106.

LIN, L. AND JIANG, Y. 2003. The computation of hitting sets: Review and new algorithms. *Information Processing Letters 86,* 4, 177 – 184.

LUSSIER, D. A. AND OLSHAVSKY, R. W. 1979. Task complexity and contingent processing in brand choice. *Journal of Consumer Research 6*, 154–165.

MAILHARRO, D. 1998. A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12*, 383–397.

MANDL, M., FELFERNIG, A., TEPPAN, E., AND SCHUBERT, M. 2011. Consumer decision making in knowledge-based recommendation. *Journal of Intelligent Information Systems 37*, 1–22.

MANNING, C. D., RAGHAVAN, P., AND SCHTZE, H. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

MARSHALL, P. H. AND WERDER, P. R. 1972. The effects of the elimination of rehearsal on primacy and recency. *Journal of Verbal Learning and Verbal Behavior 11,* 5, 649 – 653.

MAUSS, J. AND TATAR, M. M. 2002. Computing minimal conflicts for rich constraint languages. In *Proceedings of the 15th European Conference on Artificial Intelligence*, F. van Harmelen, Ed. IOS Press, 151–155.

MCDERMOTT, J. 1982. R1: A rule-based configurator of computer systems. *Artificial Intelligence 19,* 1, 39 – 88.

MCDERMOTT, J. 1994. *R1 (XCON) at age 12: lessons from an elementary school achiever*. MIT Press, Cambridge, MA, USA, 241–247.

MCGUINNESS, D. L. AND WRIGHT, J. R. 1998. An industrial strength description logics-based configurator platform. *IEEE Intelligent Systems 13*, 69–77.

MCSHERRY, D. 2004. Maximally successful relaxations of unsuccessful queries. In *15th Conference on Artificial Intelligence and Cognitive Science*. Galway, Ireland, 127–136.

MCSHERRY, D. 2005. Retrieval failure and recovery in recommender systems. *Artificial Intelligence Review 24,* 3-4, 319–338.

MILLER, R. B. 1968. Response time in man-computer conversational transactions. In *Proceedings of the Joint Computer Conference*. ACM, New York, NY, USA, 267–277.

MITTAL, S. AND FALKENHAINER, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the eighth National Conference on Artificial Intelligence*. AAAI'90. AAAI Press, 25–32.

MITTAL, S. AND FRAYMAN, F. 1989. Towards a generic model of configuraton tasks. In *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 2*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1395–1401.

MOBASHER, B. 2007. Recommender systems. In *Kunstliche Intelligenz, Special Issue on Web Mining*. Vol. 3. Bottcher IT Verlag, Bremen, Germany, 41–43.

MURTHI, B. P. S. AND SARKAR, S. 2003. The role of the management sciences in research on personalization. *Management Science 49,* 10, 1344–1362.

NGUYEN, T. A., PERKIN, W., LAFFEY, T. J., AND PECORA, D. 1987. Knowledge base verification. *AI Magazine 8*, 69–75.

O'CALLAGHAN, B., O'SULLIVAN, B., AND FREUDER, E. 2005. Generating corrective explanations for interactive constraint satisfaction. In *Principles and Practice of Constraint Programming - CP 2005*, P. van Beek, Ed. Lecture Notes in Computer Science, vol. 3709. Springer, Berlin / Heidelberg, 445–459.

O'SULLIVAN, B., PAPADOPOULOS, A., FALTINGS, B., AND PU, P. 2007. Representative explanations for over-constrained problems. In *Proceedings of the National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, Menlo Park, United States, 323–328.

PAYNE, J. W. 1976. Task complexity and contingent processing in decision making: An information search and protocol analysis. *Organizational Behavior and Human Decicion Processes 16*, 366–387.

PAZZANI, M. AND BILLSUS, D. 1997. Learning and revising user profiles: The identification of interesting web sites. *Machine Learning 27*, 313–331.

PAZZANI, M. AND BILLSUS, D. 2007. Content-based recommendation systems. In *The Adaptive Web*, P. Brusilovsky, A. Kobsa, and W. Nejdl, Eds. Lecture Notes in Computer Science, vol. 4321. Springer, Berlin / Heidelberg, 325–341.

PAZZANI, M. J. 1999. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review 13*, 393–408.

PINE, B. J. 1999. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press.

PU, P. AND CHEN, L. 2008. User-involved preference elicitation for product search and recommender systems. *AI Magazine 29,* 4, 93–103.

RAATIKAINEN, M., SOININEN, T., MNNIST, T., AND MATTILA, A. 2005. Characterizing configurable software product families and their derivation. *Software Process: Improvement and Practice 10,* 1, 41–60.

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*. ACM, New York, NY, USA, 175–186.

RICCI, F., ROKACH, L., SHAPIRA, B., AND KANTOR, P. B. 2011. *Recommender systems handbook*. Springer, New York; London.

RICCI, F., VENTURINI, A., CAVADA, D., MIRZADEH, N., BLAAS, D., AND NONES, M. 2003. Product recommendation with interactive query management and twofold similarity. In *Case-Based Reasoning Research and Development*, K. Ashley and D. Bridge, Eds. Vol. 2689. Springer, Berlin / Heidelberg, 1066–1066.

RICH, E. 1979. User modeling via stereotypes. *Cognitive Science 3*, 329–354.

SABIN, D. AND WEIGEL, R. 1998. Product configuration frameworks-a survey. *IEEE Intelligent Systems 13*, 42–49.

SALTON, G., Ed. 1988. *Automatic text processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

SALTON, G. AND BUCKLEY, C. 1988. Term-weighting approaches in automatic text retrieval. *Information Processing and Management: an International Journal 24*, 513–523.

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 REIDL, J. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*. ACM, New York, NY, USA, 285–295.

SARWAR, B., KARYPIS, G., KONSTAN, J., AND RIEDL, J. 2000. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*. ACM, New York, NY, USA, 158–167.

SARWAR, B., KARYPIS, G., KONSTAN, J., AND RIEDL, J. 2002. Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Fifth International Conference on Computer and Information Science*. 27–28.

SCHAFER, J. B., FRANKOWSKI, D., HERLOCKER, J., AND SEN, S. 2007. Collaborative filtering recommender systems. In *The adaptive web*, P. Brusilovsky, A. Kobsa, and W. Nejdl, Eds. Springer-Verlag, Berlin, Heidelberg, 291–324.

SCHLOBACH, S. AND CORNET, R. 2003. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 355–360.

SCHLOBACH, S., HUANG, Z., CORNET, R., AND VAN HARMELEN, F. 2007. Debugging incoherent terminologies. *Journal of Automated Reasoning 39,* 3, 317–349.

SCHUBERT, M. 2009. Personalized query relaxations and repairs in knowledge-based recommendation. In *Proceedings of the 2009 ACM Conference on Recommender Systems*. ACM, 409–412.

SCHUBERT, M. AND FELFERNIG, A. 2011. Bfx: Diagnosing conflicting requirements in constraint-based recommendation. *International Journal on Artificial Intelligence Tools 20,* 2, 297–312.

SCHUBERT, M., FELFERNIG, A., AND MANDL, M. 2009. Solving over-constrained problems using network analysis. In *Proceedings of the 2009 International Conference on Adaptive and Intelligent Systems*. Klagenfurt, Austria, 9–14.

SCHUBERT, M., FELFERNIG, A., AND MANDL, M. 2010. Fastxplain: Conflict detection for constraint based recommendation problems. In *Trends in Applied Intelligent Systems - 23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems*. Lecture Notes in Computer Science. Springer, 621–630.

SCHUBERT, M., FELFERNIG, A., AND REINFRANK, F. 2011. Reaction: Personalized minimal repair adaptations for customer requests. In *Proceedings of the 9th International Conference of Flexible Query Answering Systems*. Lecture Notes in Computer Science, vol. 7022. Springer Berlin / Heidelberg, 13–24.

SHARDANAND, U. AND MAES, P. 1995. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of Conference of Human Factors in Computing Systems*. ACM Press, 210–217.

SIDDIQI, S. AND HUANG, J. 2007. Hierarchical diagnosis of multiple faults. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, M. M. Veloso, Ed. 581–586.

SILVA, J. P. M. AND SAKALLAH, K. A. 1996. Grasp: a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD 96. IEEE Computer Society, Washington, DC, USA, 220–227.

SILVEIRA, D. B. G. D. AND FOGLIATTO, F. S. 2001. Mass customization: Literature review and research directions. *International Journal of Production Economics 72*, 1–13.

SINZ, C. AND HAAG, A. 2007. Configuration. *IEEE Intelligent Systems 22,* 1, 78–90.

SMYTH, B., BALFE, E., BOYDELL, O., BRADLEY, K., BRIGGS, P., COYLE, M., AND FREYNE, J. 2005. A live-user evaluation of collaborative web search. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1419–1424.

SOININEN, T. AND STUMPTNER, M. 2003. Special issue: Configuration.

SOININEN, T., TIIHONEN, J., MÄNNISTÖ, T., AND SULONEN, R. 1998. Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AIEDAM) 12*, 357–372.

STRUSS, P. 2002. Model-based diagnosis for industrial applications. *IEEE Colloquium on Applications of Model-Based Reasoning*, 1–9.

STUMPNER, M., HASELBOCK, A., AND FRIEDRICH, G. 1994. Cocos - a tool for constraint-based, dynamic configuration. In *Proceedings of the 10th Conference on Artificial Intelligence for Applications*. San Antonia, TX, USA, 373–380.

STUMPTNER, M. 1997. An overview of knowledge-based configuration. *AI Communications 10*, 111–125.

THOMPSON, C. A., GÖKER, M. H., AND LANGLEY, P. 2004. A personalized system for conversational recommendations. *Journal of Artificial Intelligence Research 21,* 1 (March), 393–428.

TIIHONEN, J., SOININEN, T., NIEMEL, I., AND SULONEN, R. 2003. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design (ICED 03)*. 1290–1299.

TSANG, E. 1993. *Foundations of Constraint Satisfaction*.

TSENG, M. M. AND JIAO, J. 2001. *Handbook of Industrial Engineering, Technology and Operation Management*, 3rd edition ed. Wiley, New York, NY, USA, Chapter Mass Customization.

TVERSKY, A. AND KAHNEMAN, D. 1981. The framing of decisions and the psychology of choice. *Science, New Series 211*, 453–458.

VON DER MAEN, T. AND LICHTER, H. 2004. Requiline: A requirements engineering tool for software product lines. In *Software Product-Family Engineering*. Lecture Notes in Computer Science, vol. 3014. Springer Berlin / Heidelberg, 168–180.

VON WINTERFELDT, D. AND EDWARDS, W. 1986. *Decision Analysis and Behavioral Research*. Cambridge University Press.

WASSERMAN, S. AND FAUST, K. 1994. *Social Network Analysis*. Cambridge University Press, Cambridge.

WILSON, D. R. AND MARTINEZ, T. R. 1997. Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research (JAIR) 6*, 1–34.

WOTAWA, F. 2001. A variant of reiter's hitting-set algorithm. *Information Processing Letters 79*, 45–51.

WRIGHT, J. R., WEIXELBAUM, E. S., VESONDER, G. T., BROWN, K. E., PALMER, S. R., BERMAN, J. I., AND MOORE, H. H. 1993. A knowledge-based configurator that supports sales, engineering, and manufacturing at at&t network systems. *AI Magazine 14,* 3, 69–80.

YONGLI, Z., LIMIN, H., AND JINLING, L. 2006. Bayesian networks-based approach for power systems fault diagnosis. *IEEE Transaction on Power Delivery 21*, 634–639.

YU, B. AND SKOVGAARD, H. J. 1998. A configuration tool to increase product competitiveness. *IEEE Intelligent Systems 13*, 34–41.

ZANKER, M. AND JESSENITSCHNIG, M. 2009. Collaborative feature-combination recommender exploiting explicit and implicit user feedback. In *Proceedings of the 2009 IEEE Conference on Commerce and Enterprise Computing*. IEEE Computer Society, Washington, DC, USA, 49–56.

ZANKER, M., JESSENITSCHNIG, M., JANNACH, D., AND GORDEA, S. 2007. Comparing recommendation strategies in a commercial context. *IEEE Intelligent Systems 22*, 69–73.

ZANKER, M., JESSENITSCHNIG, M., AND SCHMID, W. 2010. Preference reasoning with soft constraints in constraint-based recommender systems. *Constraints 15*, 574–595.

Appendix

Further *WeCoTin* Evaluation Results

This section presents further evaluation results for the algorithms presented in Chapter 4 using *WeCoTin* Models. The *WeCoTin* Models have already been described in Section 4.5.2. In this Section the evaluation results of different models are presented. Although the evaluation results are similar to the ones described in Section 4.5.2, the presented results can give a further insight to the performance of the algorithms. The models that have been used for the evaluation presented in this section differ in terms of number of rules and number of variables. An overview of the models used is given in Table 7.1.

Table 7.1.: Overview of the *WeCoTin* configuration knowledge bases that are used for the run time performance tests

	Number of Rules	Number of Variables
Test Case 1	10	34
Test Case 2	13	28
Test Case 3	17	31
Test Case 4	14	24
Test Case 5	23	20
Test Case 6	14	23
Test Case 7	13	28
Test Case 8	5	144
Test Case 9	4	11
Test Case 10	84	242

For the evaluations presented in this section, all algorithms have been applied to different models with 10 and 20 generated requirements. These requirements have been generated 100 times for each model in a way that they are inconsistent with the tested model. For each of the 200 settings (100 settings for 10

requirements and 100 settings for 20 requirements), the number of needed solver calls (consistency checks) to calculate one and all diagnoses has been measured.

Table 7.2.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 1* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	20	53	16	38
FastDiag	11	33	9	28
PersDiag	20	53	16	38
CorrectiveRelax	8	32	9	27
FlexDiag (m=2)	5	62	8	256
FlexDiag (m=3)	5	70	6	751

Table 7.2 shows that the algorithms *FlexDiag (m=2)* and *FlexDiag (m=3)* scale badly when calculating all diagnoses. This can especially seen from the results with 20 requirements. Nevertheless, both algorithms (*FlexDiag (m=2)* and *FlexDiag (m=3)*) perform very well when only one diagnosis is calculated. This is based on the fact that both algorithms aim to calculate diagnoses. Compared to the *FlexDiag* algorithms ($m = 2$ and $m = 3$), the *FastDiag* algorithm performs a bit worse for calculating one diagnosis. This can be explained by the fact, that the *FlexDiag* algorithms ($m = 2$ and $m = 3$) calculate only diagnosis clusters, whereas the *FastDiag* algorithm calculates minimal diagnoses.

Table 7.3.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 2* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	19	31	50	117
FastDiag	9	26	14	155
PersDiag	19	31	46	127
CorrectiveRelax	16	35	21	207
FlexDiag (m=2)	5	89	10	517
FlexDiag (m=3)	3	154	6	674

Another evaluation result is presented in Table 7.3. From this table, it can be seen that *QuickXplain* algorithm and the *PersDiag* algorithm have a similar performance. The reason for this is that both algorithms

use the same approach to calculate minimal conflict sets. The main difference of these two approaches is that the *PersDiag* algorithm integrates personalisation strategies to identify personalized diagnoses. Nevertheless, this integration has not much effect on the number of consistency checks (see Table 7.3). Only for the settings with 20 requirements, the results differ. The *QuickXplain* algorithm, for example, needs more consistency checks for calculating one minimal diagnosis due to the breadth-first search in the Hitting Set Directed Acyclic Graph (HSDAG) (Reiter, 1987). Compared to this, the *PersDiag* algorithm uses a best-first search in the HSDAG (the criteria for the best node comes from the personalisation strategy). This best-first search criteria can also result in more consistency checks, if all diagnoses are calculated.

Table 7.4.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 3* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	35	48	77	111
FastDiag	11	58	16	65
PersDiag	35	48	73	119
CorrectiveRelax	16	71	19	77
FlexDiag (m=2)	8	76	11	123
FlexDiag (m=3)	6	77	7	272

Table 7.5.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 4* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	15	16	6	6
FastDiag	4	12	5	6
PersDiag	15	16	6	6
CorrectiveRelax	8	19	9	10
FlexDiag (m=2)	5	57	4	35
FlexDiag (m=3)	4	76	3	65

Table 7.4 presents the outcomes of the evaluation performed with another model. The size of this model is similar to the ones presented in Table 7.2 and 7.3. Nevertheless, the run time performance varies especially when calculating all diagnoses for 20 customer requirements. This fact can be seen very well, when the number of consistency checks is compared for the *FlexDiag* algorithms. In Table 7.2 the number

of consistency checks is about 10 times (for *FlexDiag (m=2)*) larger compared to the ones needed by the other algorithms. This is not the case in Table 7.3 nor in Table 7.4. The reason for this behaviour is that the performance of the *FlexDiag* algorithms (*m = 2* and *m = 3*) is highly dependent on the size and the number of diagnoses (for a detailed discussion see Section 4.5.3).

The interesting point about the results presented in Table 7.5, is that the number of consistency checks is lower for the settings with 20 requirements compared to the settings with 10 requirements. This happens when the configuration rules (constraints in the knowledge base) affect only few requirements. As a matter of fact, if such a requirement is included in the total set of requirements, it is responsible for the number of consistency check.

Table 7.6.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 5* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	11	12	12	13
FastDiag	6	13	4	12
PersDiag	11	12	12	13
CorrectiveRelax	9	22	8	22
FlexDiag (m=2)	5	31	6	43
FlexDiag (m=3)	3	71	4	106

Table 7.7.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 6* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	15	16	16	27
FastDiag	5	12	9	18
PersDiag	15	16	16	27
CorrectiveRelax	5	13	13	37
FlexDiag (m=2)	5	37	5	116
FlexDiag (m=3)	5	77	4	178

The outcomes of another evaluation is presented in Table 7.6. Comparing the results of the *FastDiag* algorithm and the *CorrectiveRelax* algorithm, it can be seen that *FastDiag* needs less consistency checks. For

explaining this observation, the idea of the two algorithms is pointed out again. The *CorrectiveRelax* algorithm (O'Callaghan et al., 2005) calculates corrective explanations based on a binary search. During this search the algorithm removes, one-by-one, constraints that cause inconsistency (for a detailed explanation see, for example, Section 3.8). Compared to this, the *FastDiag* algorithm performs a divide-and-conquer principle to strategically eliminate consistent constraints in order to identify the diagnosis. Using this divide-and-conquer principle, double checks on the same set of requirements are avoided. Nevertheless, these double checks may occur when using the binary search (as the *CorrectiveRelax* algorithm does).

The difference in the run time behaviour between the *PersDiag* algorithm and the *CorrectiveRelax* algorithm (O'Callaghan et al., 2005) for calculating one minimal diagnosis (see, for example, Table 7.7), can be explained by the different approaches to calculate this diagnosis. The *PersDiag* algorithm uses minimal conflict sets and the Hitting Set Directed Acyclic Graph (HSDAG) (Reiter, 1987) to determine a personalized diagnosis. This causes the higher number of consistency checks, because it is more costly to calculate minimal conflict sets first and then add them to the HSDAG to determine a diagnosis, compared to the possibility to directly calculate a diagnosis. Nevertheless, this different approach to identify diagnoses is not essential when calculating all minimal diagnoses. For example, when calculating all minimal diagnoses for 10 requirements, the *CorrectiveRelax* algorithm performs better, whereas for calculating all minimal diagnoses for 20 requirements, the *PersDiag* algorithm performs better (see, for example, Table 7.7 and 7.8).

Table 7.8.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 7* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	13	15	29	31
FastDiag	9	11	16	25
PersDiag	13	15	28	29
CorrectiveRelax	12	14	21	31
FlexDiag (m=2)	8	89	10	169
FlexDiag (m=3)	5	197	7	172

The setting used for the evaluation presented in Table 7.9 is especially interesting, because it includes a high number of variables (**144**) compared to a low number of rules (**5**). This fact and a higher structural complexity of the settings increased the number of consistency checks needed by each algorithm. For example, the *FlexDiag (m=3)* needs nearly 600 consistency checks to determine all diagnoses for 20 requirements. Nevertheless, it calculates the first diagnosis with 13 consistency checks on an average.

Table 7.9.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 8* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	83	92	57	287
FastDiag	17	49	26	124
PersDiag	59	109	38	428
CorrectiveRelax	24	62	32	170
FlexDiag (m=2)	10	71	14	240
FlexDiag (m=3)	6	60	13	567

Table 7.10.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 9* dataset to calculate one and all minimal diagnoses with 10 requirements using different algorithms for the consistency management

	10 requirements	
	1 Diagnosis	All Diagnoses
QuickXplain	24	33
FastDiag	10	30
PersDiag	23	33
CorrectiveRelax	18	40
FlexDiag (m=2)	5	41
FlexDiag (m=3)	4	44

Table 7.10 (*Test Case 9*) presents the evaluation results of the setting with the lowest number of variables (**11**) and **4** configuration rules (constraints in the knowledge base). Compared to this, Table 7.11 (*Test Case 10*) shows the evaluation results of the setting with the highest number of variables (**242**) and **84** configuration rules. As the test setting of *Test Case 9* incorporates only **11** variables, it is only possible to generate (at most) **11** requirements, due to the restriction that each variable can be assigned only once in the set of requirements. For this reason, only settings with **10** requirements have been generated. What can be seen from Table 7.10 is that the *FlexDiag* algorithms (*m = 2* and *m = 3*) perform well when calculating one diagnosis. Moreover, the algorithms *QuickXplain* and *PersDiag* have a similar performance. Another observation is that the *FastDiag* algorithm needs less consistency checks for calculating one minimal diagnosis compared to the other algorithms (note that the diagnoses calculated by the *FlexDiag* algorithms are not minimal). The results of the setting with the largest configuration knowledge base (presented in Table 7.11) lead to the same observations.

Table 7.11.: Run time performance evaluation (number of needed solver calls) of the *WeCoTin - Test Case 10* dataset to calculate one and all minimal diagnoses with 10 and 20 requirements using different algorithms for the consistency management

	10 requirements		20 requirements	
	1 Diagnosis	All Diagnoses	1 Diagnosis	All Diagnoses
QuickXplain	14	16	26	40
FastDiag	8	10	13	33
PersDiag	13	16	25	40
CorrectiveRelax	12	14	27	54
FlexDiag (m=2)	4	6	8	89
FlexDiag (m=3)	4	6	8	205