Intelligent Supporting Techniques for the Maintenance of Constraint-based Configuration Systems¹²

Florian Reinfrank, Gerald Ninaus, Franz Wotawa, Alexander Felfernig Institute for Software Technology Graz University of Technology Inffeldgasse 16b/II, 8010 Graz, Austria {firstname.lastname}@ist.tugraz.at

Abstract. Constraint-based systems like knowledge-based recommendation and configuration are well established technologies in many different product areas like cars, computers, notebooks, and financial services. Such systems reduce the number of valid products and configurations regarding the customers' preferences. The relationship between product variables and customer questions is represented by constraints. Nowadays, constraint-based configuration systems represent volatile product assortments: Variables must be adapted, new product features lead to new questions for the customer, and / or the constraints must be updated. We call such scenarios maintenance tasks.

In complex constraint-based configuration systems the maintenance task is time consuming and error prone. Previous research focused on the detection of conflicts, repair actions for the conflicts, and redundant constraints. In this paper we give an overview about these techniques and present new approaches like recommendation, well-formedness violation, simulation, and knowledge base verification for the support of knowledge engineers.

1 Introduction

In e-Commerce applications constraint-based configuration systems are used to show which combinations of product variable assignments can be combined and offered to potential customers. Due to complex restrictions to adapt the product assortment regarding the customers' preferences, intelligent techniques can be used if the customers' preferences can not be fulfilled.

A knowledge engineer develops and maintains such knowledge bases. Based on knowledge (for example, knowledge of bikes) from domain experts, the engineer defines product variables and variable domains (e.g., the domain of the product variable BikeType' is MountainBike, CityBike, and RacerBike), prepares additional questions presented to potential customers (e.g., 'What is the main usage of the bike?'), and develops relationships (constraints) between questions and products (e.g., if the main usage of the bike is $everyday_life$ then the bike should be of the type CityBike).

Such knowledge bases must be updated over time. For example, in the last years bikes with an electric engine became popular.

The knowledge engineer has to extend the current knowledge base with new product attributes (e.g., introducing a new product feature eBike) and questions (e.g., 'Do you want an electric engine assistance?'). In complex constraint-based configuration system updates are time consuming and error prone because unexperienced knowledge engineers have to adapt the knowledge base and unexpected dependencies between constraints exist.

In this paper we show how we can support knowledge engineers when they maintain a constraint-based configuration system. The approaches can be used in many scenarios like knowledge-based recommendation, knowledge-based configuration, or feature models. Due to complex restrictions to adapt the product assortment regarding the customers' preferences, intelligent techniques can be used if the preferences can not be fulfilled.

This paper is organized as follows. Section 2 gives an overview about constraint-based configuration systems. It introduces a running example and relevant definitions for this paper. Our new approaches to support knowledge engineers in maintaining constraint-based configuration systems are described in Section 3. A summary in Section 4 concludes this paper.

2 Related Work

In this Section we give an overview about constraint-based configuration systems, introduce a running example for this paper and define relevant terms which are necessary to explain the intelligent supporting techniques from Section 3.

For our constraint-based configuration system we use the constraint satisfaction problem (CSP) modeling technique [14]. A CSP is a triple (V, D, C) and consists of a set of variables V and a set of domains D where each domain $dom(v_i)$ represents all valid assignments for a variable v_i , e.g., $dom(v_i) = \{val_1, \dots, val_n\}$. The set C contains all constraints which restrict the number of valid instances of a constraint-based configuration system. Basically, a constraint consists of a set of assignments a for variables and relations between them. The set $A(c_i)$ is the set of assignments a constraint c_i has. If a constraint contains only one assignment, we denote such constraints unary constraint or assignment [13]. Furthermore, the constraints can be divided into two different types. First, the set C_{KB} contains all constraints which describe the domain. For example, it is not allowed to use mountain bike tires (TireWidth > 50mm) for racing bikes (BikeType = RacerBike), s.t. $c = \neg(BikeType =$ $RacerBike \wedge TireWidth > 50mm$). Second, the committed cus-

 $^{^1}$ We thank the anonymous reviewers for their helpful comments.

² The work presented in this paper has been conducted within the scope of the research project ICONE (Intelligent Assistance for Configuration Knowledge Base Development and Maintenance) funded by the Austrian Research Promotion Agency (827587).

tomers' preferences are represented as constraints in the set C_R .

The following example is denoted as a CSP and shows a bike knowledge base. It contains variables which represent product variables as well as customer requirements. The set C_R contains an example for customer requirements.

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

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

$$C_{KB} = \{ C_{KB} = \{ c_0 := BikeType = MountainBike \rightarrow TireWidth > 37mm \land FrameSize \ge 50cm; \\ c_1 := BikeType = RacerBike \rightarrow TireWidth = 23mm \land FrameSize = 60cm; \\ c_2 := BikeType = CityBike \rightarrow TireWidth = 37mm \land FrameSize \ge 50cm; \\ c_3 := \neg (BikeType \ne CityBike \land eBike = true); \\ c_4 := Usage = EverydayLife \rightarrow BikeType = CityBike; \\ c_5 := Usage = HillClimbing \rightarrow BikeType = MountainBike; \\ c_6 := Usage = Competition \rightarrow BikeType = RacerBike \land FrameSize = 60cm; \\ c_7 := eBike = true \rightarrow TireWidth = 37mm; \\ c_8 := UniCycle = false; \\ \} \\ C_R = \{ c_9 : FrameSize = 50cm; \\ c_{10} : Usage = Competition; \\ c_{11} : eBike = true; \\ \} \\ C = C_{KB} \cup C_R \end{cases}$$

The example contains some anomalies in terms of conflicts, redundancies and well-formedness violations. Figure 1 gives an overview of different types of anomalies. In the following, we list definitions to define the anomalies.

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

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

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

Definition 2 'Complete Instance': An instance is complete iff all product variables have an assignment, such that $\forall_{v \in V} v \neq \emptyset$.

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



Figure 1. Different types of anomalies.

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

In constraint-based configuration systems it can happen, that the system can not offer consistent instances to a user (anomaly) because it is not possible to satisfy all constraints (see Definition 3). Such a 'no solution could be found' dilemma is caused by at least one conflict between a) the constraints in the knowledge base C_{KB} and the customer requirements C_R or b) within the set C_{KB} itself. Definition 4 introduces a formal representation of a conflict.

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

If we have an inconsistency in our knowledge base, we can say that $C_{KB} \cup C_R$ is always a conflict set. To have a more detailed information about the inconsistency, we introduce the term 'minimal conflict' which is described in Definition 5.

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

When we focus on the set C_R and say, that C_{KB} is consistent, our example contains two minimal conflict sets. $CS_1 = \{c_9, c_{10}\}$ because it is not possible to have a bike for competition with a frame size of 50cm and $CS_2 = \{c_{10}, c_{11}\}$ because bikes used for *competition* do not support eBikes. The example shows that a knowledge base can have more than one conflict. In such cases we can help users to resolve the conflicts with diagnosis. A diagnosis Δ is a set of constraints. The removal of the set Δ from C_R leads to a consistent knowledge base, formally described in Definition 6.

Definition 6 '*Diagnosis*': A diagnosis Δ is a set of constraints $\Delta \subseteq$ $C_R \cup C_{KB}$. When removing the set Δ from $C_R \cup C_{KB}$, the knowledge base will be consistent, s.t. $C_R \cup C_{KB} \setminus \Delta$ is consistent.

Assuming that C_{KB} is consistent (see Definition 3), we can say that the knowledge base always will be consistent if we remove C_R . In Definition 7 we introduce the term 'minimal diagnosis' which helps to reduce the number of constraints within a diagnosis.

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

Juha Tiihonen, Andreas Falkner and Tomas Axling, Editors Proceedings of the 17th International Configuration Workshop September 10-11, 2015, Vienna, Austria

The example configuration knowledge base contains two minimal diagnoses. The removal of the set $\Delta_1 = \{c_9, c_{11}\}$ or $\Delta_2 = \{c_{10}\}$ leads to a consistent configuration knowledge base. After having calculated diagnoses and removed the constraints which are in one diagnosis, we can ensure a consistent knowledge base which is necessary for calculating redundancies and well-formedness violations.

A redundancy is a set of redundant constraints in the knowledge base. A constraint c is redundant, if a knowledge base KB' without the constraint c has the same semantics³ as the knowledge base KB which contains the constraint. Redundant constraints are formally described in Definition 8.

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

In our example, the constraint c_7 is redundant, since only CityBikes can be eBikes (c_3) and have tires with a width of 37mm (c_2).

While conflicts, diagnoses, and redundancies focus on constraints, well-formedness violations identify anomalies based on variables and domain elements [2]. We now introduce well-formedness violations in constraint-based configuration systems.

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

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

The assignments FrimeSize = 40cm and UniCycle = true; can never be part of a consistent instance because MountainBikesand CityBikes require at least 50cm and RacerBikes require a FrameSize of 60cm and our current knowledge base does not support UniCycles.

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

Definition 10 'Full mandatory': A domain element $val_1 \in dom(v_i)$ is full mandatory iff there is no consistent (complete or incomplete) instance where the variable v_i does not have the assignment val_1 , s.t. $C_{KB} \cup \{v_i \neq val_1\}$ is inconsistent.

The knowledge base can never be consistent if $UniCycle \neq false$. In that case, we can say that the domain element false of the domain dom(UniCycle) is full mandatory and UniCycle = true can never be in a consistent knowledge base (dead domain element). Another well-formedness violation is called unnecessary refinement. Such an unnecessary refinement consists of two variables. If the first variable has an assignment, it is possible to predict the assignment of the second variable because the second variable can only have exactly one consistent assignment. A formal definition is given in Definition 11.

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

In our example the variable pair Usage and BikeType is unnecessary refined because whenever Usage = EverydayLife the BikeType = CityBike, Usage = HillClimbing always leads to BikeType = MountainBike, and Usage = Competition is always combined with the assignment BikeType = RacerBike. If such a violation occurs, we can recommend the knowledge engineer to remove the variable Usage and replace it with the variable BikeType in the constraints.

3 Intelligent Support for the Maintenance of Constraint-based configuration systems

In this Section we describe existing (conflict and redundancy management) and new (recommendation, well-formedness, simulation, metrics) intelligent techniques to support knowledge engineers in their maintenance tasks.

3.1 Intelligent Recommendation

Constraint-based knowledge bases can have hundreds or thousands of variables, domain elements, and constraints. If there is a maintenance task (e.g., inserting new tire sizes), recommendation techniques help to differentiate between relevant and not relevant information within the knowledge base. For example, the tires of a bike probably have an influence on the frame of a bike but does not influence the bell of a bike. In such cases, recommendation techniques detect items (variables, domain elements, constraints, test cases) which are influenced by the tires and the knowledge engineer can focus on these items. We describe four different types of recommendation to support knowledge engineers in their maintenance tasks [4].

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

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

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

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

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

In table 1 we try to find out if we should recommend item c_7 to knowledge engineer u_0 . The common process to find recommendable items is twofold. First, we try to find knowledge engineers with similar interests. In our example, u_1 and u_2 have nearly the same items viewed or edited. Second, we have to evaluate if the similar

 $[\]frac{3}{3}$ We use the term '*semantics*' to describe a knowledge base KB' with the same solution set as KB.

knowledge engineers are interested in the item. Therefore, we use the Pearson correlation [3, 6]. In our example, u_1 , and u_2 have viewed / edited item c_7 and we can recommend c_7 to knowledge engineer c_0 . Another recommendation approach is the usage of *content-based filtering*. The basic idea is to find similar items compared to a reference item. We take the variable names and domain values from a constraint and evaluate the similarities between the reference item and all other items. The similarities are measured by the TF-IDF (term frequency and inverse document frequency) algorithm [6, 8] where the item is the document and the terms are the variables and domain elements. Table 2 shows the similarity values between constraint c_7 as reference constraint with the other constraints.

constraint	similarity
c_0	0.50
c_1	0.17
c_2	0.50
c_3	0.00
c_4	0.00
c_5	0.00
c_6	0.33
C8	0.00

Table 2. Similarities between constraint c_7 with the other constraints based on content based recommendation

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

3.2 Intelligent Anomaly Management

As mentioned in Section 2, there are many anomalies in our example knowledge base. In the following, we describe algorithms for detecting conflicts, diagnoses, redundancies, and well-formedness violations as well as explanations for those anomalies. These algorithms reduce the time to detect the anomalies and explain the anomaly to get a higher understanding of the knowledge base.

Junker [7] described a divide-and-conquer approach to detect conflicts in knowledge bases. The algorithm takes two sets of constraints as input. C_{KB} is a set of constraints which can not be part of a diagnosis. The constraints in the set C_R will be taken into account for calculating a diagnosis. If the set C_R is not empty and $C_R \cup C_{KB}$ is not consistent, the algorithm returns a set of constraints which is a minimal conflict (see Definition 5).

Algorithm 1 QuickXI	Plain (C_{KB}, C_R) : Δ
	$\triangleright C_{KB}$: set of not diagnosable constraints
	$\triangleright C_R$: set of diagnosed constraints
if $isEmpty(C_R)$ or $return \emptyset;$	$r \ consistent(C_{KB} \cup C_R)$ then
else return Quick2	$XPlain'(C_{KB}, \Delta, C_R);$
end if	

The algorithm QuickXPlain' takes three sets as input. While Δ is initially empty, the set C_{KB} contains the constraints which can not be part of a conflict and the constraints which are part of a conflict are in the set C_R . Note that the set C_{KB} can also be empty. The algorithm is a recursive divide-and-conquer algorithm. It splits the set C_R into two parts (C_1 and C_2) and adds the part C_1 to C_{KB} . C_2 will be evaluated by doing a recursive call with C_2 as the set which has to be evaluated.

Contrary to QuickXPlain, FastDiag is an algorithm to calculate a minimal diagnosis and has C and C_R as input. C contains all constraints, s.t. $C = C_{KB} \cup C_R$. If C is an empty set, FastDiag has no diagnosable set and the algorithm stops. It also stops if the set $C \setminus C_R$ is inconsistent, because this set contains inconsistencies, but will not be diagnosed. If both preconditions are fulfilled, Algorithm 4 will calculate one diagnosis.

Algorithm 3 FASTDIAG	$G(C_R, C):\Delta$
$\triangleright C_F$	R: Set of constraints which will be diagnosed
$\triangleright C$: inconsist	ent knowledge base including all constraints
if $C = \emptyset \lor inconsis$	tent(CKB - C) then
$return \ \emptyset;$	
else	
$return DIAG(\emptyset$	(C_R, C)
end if	

First of all, DIAG checks whether C is consistent. If it is consistent, each subset of C is also consistent and no constraint in C can be a part of the diagnosis. Otherwise, C_R will be divided into two subsets C_1 and C_2 . Each subset will be removed from C separately and checked again in a recursive manner. If $C \setminus C_1$ is consistent, we can say that C_2 is consistent and an empty set will be returned. If it is inconsistent, at least one constraint in C_1 must be part of the diagnosis and therefore C_1 will be divided and tested again unless |C| = 1. The algorithm returns $\Delta_1 \cup \Delta_2$ which is a minimal diagnosis.

With the previous algorithms we can a.) support customers when they do not get any products for their preferences ($C_{KB} \cup C_R$ is inconsistent) and b.) support knowledge engineers when they maintain a constraint-based configuration system with conflicts in C_{KB} . For a detailed description of the visualization of conflicts we refer the reader to [16].

When we can assume that C_{KB} is consistent, we can continue with redundancy and well-formedness checks. Please note that the following algorithms are applied to the constraint set C_{KB} and ignore C_R , s.t. $C = C_{KB}$.

The first approach for detecting redundancies has been proposed by Piette [9]. The approach is the following: a knowledge base aggregated with its negotiation must be inconsistent, formally described as $C \cup \overline{C}$ is inconsistent and $\overline{C} = \{\neg c_0 \lor \neg c_1 \lor ... \lor \neg c_n\}$. By removing a constraint c_i separately from C, the algorithm checks whether the result of $C - \{c_i\} \cup \overline{C}$ is still inconsistent. If this is the case, then the constraint c_i is redundant and can be removed. Finally, the algorithm

Algorithm 4 DIAG(Δ , C_R , C): Δ

 $\triangleright \Delta: \text{ Set of diagnosed constraints} \\ \triangleright C_R: \text{ Set of constraints which will be diagnosed} \\ \triangleright C: C_R \cup C_{KB} \\ \text{if } \Delta \neq \emptyset \text{ and consistent}(C) \text{ then} \\ return \, \emptyset; \\ \text{end if} \\ \text{if singleton}(C_R) \text{ then} \\ return \, C_R; \\ \text{end if} \\ k \leftarrow \lceil \frac{|C_R|}{2} \rceil; \\ C_1 \leftarrow \{c_0, ..., c_k\} \in C_R; \\ C_2 \leftarrow \{c_{k+1}, ..., c_n\} \in C_R; \\ \Delta_1 \leftarrow DIAG(C_1, C_2, C - C_1); \\ \Delta_2 \leftarrow DIAG(\Delta_1, C_1, C - \Delta_1); \\ return(\Delta_1 \cup \Delta_2); \\ \end{cases}$

returns the set C without redundant constraints.

Algorithm 5 SEQUENTIAL(C): Δ

 $\succ C: \text{ knowledge base}$ $\succ \overline{C}: \text{ the complement of } C$ $\succ \Delta: \text{ set of redundant constraints}$ $C_t \leftarrow C;$ for all $c_i \text{ in } C_t \text{ do}$ if $isInconsistent(C_t - c_i \cup \overline{C})$ then $C_t \leftarrow C_t - \{c_i\};$ end if
end for $\Delta \leftarrow C - C_t;$ $return \Delta;$

Another approach (CoreDiag) has been proposed by Felfernig et al. [5]. Instead of a linear approach, they adapt the QuickXPlain algorithm. The divide-and-conquer approach of this algorithm checks whether removing a set of constraints C_1 leads to an inconsistency formally described as $C - C_1 \cup \overline{C}$ is inconsistent. If it is not inconsistent, C_1 must be further divided and tested again.

Algorithm 6 COREDIAG (CKB): Δ	
	$\triangleright C$: set with all constraints
	$\triangleright \overline{C}$: the complement of C
\triangleright 2	∆: set of redundant constraints
$\overline{C} \leftarrow \{\neg c_1 \lor \neg c_2 \lor \dots \lor \neg c_n\};\\ return(C - \text{CORED}(\overline{C}, \overline{C}, C));$	

CoreD (Algorithm 7) checks, if $B \subseteq C$ is inconsistent. An inconsistency of $B \cup \overline{C}$ means that the subset is not redundant and no constraint of B will be a part of Δ . singleton(C) = true means that |C| is redundant and will be returned. Otherwise the constraint set C will be further divided and the subsets will be checked recursively. With the presented approaches we can calculate one conflict, diagnosis, or constraint set without redundancies. In complex knowledge bases we can assume that many anomalies are in the knowledge base. For calculating all conflicts / diagnoses / redundant constraint sets, we use Reiter's HSDAG approach [12]. This approach takes the result of one of the algorithms above and expands branches for each constraint in the result set. The constraint will be inserted into the set which can not be part of the result, e.g. a constraint c_i will be removed from C_R and added to C_{KB} in the QuickXPlain algorithm.

Algorithm 7 CORED (B, Δ, C) : Δ \triangleright B: Consideration set $\triangleright \Delta$: Constraints added to B \triangleright C: set of constraints to be checked for redundancy if $\Delta \neq \emptyset$ and inconsistent(B) then return \emptyset ; end if if singleton(C) then return(C);end if $k \leftarrow \left\lceil \frac{|C|}{2} \right\rceil;$ $C_1 \leftarrow \{\tilde{c}_1, c_2, \dots, c_k\} \in C;$ $C_2 \leftarrow \{c_{k+1}, c_{k+2}, ..., c_n\} \in C;$ $\Delta_1 \leftarrow \text{CORED}(B \cup C_2, C_2, C_1);$ $\Delta_2 \leftarrow \text{CORED}(B \cup \Delta_1, \Delta_1, C_2);$ return $(\Delta_1 \cup \Delta_2);$

Hence the shifted constraint can not be part of an anomaly and further anomalies can be detected.

Both algorithms (SEQUENTIAL and CoreDiag) can be used to detect redundant constraints. As mentioned in Section 2 a constraint consists of a set of variable assignments $A(c_i)$. When we want to test if an assignment of a constraint is redundant, we have to remove the assignment from the constraint and check, if the knowledge base is still redundant. For a detailed description of assignment-based redundancy detection we refer the reader to [11].

We also have to discuss the usefulness of redundant constraints. On the one hand, desired redundancies can help to increase the understanding of a knowledge base. For example, if many implications (e.g. $A \rightarrow B$; $B \rightarrow C$; $C \rightarrow D$) are in the knowledge base, a constraint $A \rightarrow D$ may helps to understand the knowledge base. On the other hand, redundant constraints can increase the effort for updates. If the redundant constraint are not identified by the knowledge engineer, the knowledge base does not have a correct behavior anymore. Next, we describe the algorithms to detect well-formedness violations. First, Algorithm 8 takes sets of constraints (C) and variables (V) as input parameters and returns a set of variable assignments. Each of the assignments can never be consistent with C. The suggestion for the knowledge engineer is, that the domain elements which will be returned by the algorithm can be deleted.

Algorithm 8 DeadDomainElement (C, V) : Δ
\triangleright C: knowledge base constraints
\triangleright V: knowledge base variables
$\triangleright \Delta$ set with inconsistent variable assignments
for all $v_i \in V$ do
for all $dom_j \in dom(v_i)$ do
$C' = C \cup \{v_i = dom_j\}$
if $inconsistent(C')$ then
$\Delta \leftarrow \{v_i = dom_j\}$
end if
end for
end forreturn Δ

While we can evaluate if a domain element can never be in a consistent instance, we can also check if a domain element must be in a consistent instance of a knowledge base. We denote such domain elements as *full mandatory*. Algorithm 9 checks whether the knowledge base will be inconsistent, if the domain element dom_j is not selected.

Algorithm 9 FullMandatory (C, V) : Δ
\triangleright C: knowledge base constraints
\triangleright V: knowledge base variables
$\triangleright \Delta$ set with inconsistent variable assignments
for all $v_i \in V$ do
for all $dom_j \in dom(v_i)$ do
$C' = C \cup \{v_i \neq dom_j\}$
if $inconsistent(C')$ then
$\Delta \leftarrow \{v_i \neq dom_j\}$
end if
end for
end forreturn Δ

If variable v_i contains a full mandatory domain element, we can say, that each other domain element of v_i is a dead element. If a domain element is full mandatory, we suggest the knowledge engineer to delete all other domain elements or to remove the variable itself. Finally, we introduce an algorithm to detect unnecessary refinements between variables (see Definition 11). Algorithm 10 returns a set of constraints. Each of these constraints describe one unnecessary refinement between two variables and each domain element between both variables. The assignments between the variables are conjunctive and each domain element of variable v_i is in a disjunctive order, e.g. $(v_i = val_{i1} \land v_j = val_{j1}) \lor (v_i = val_{i2} \land v_j = val_{j2}) \lor (v_i = val_{i3} \land v_j = val_{j3})$.

Algorithm 10 UnnecessaryRefinement (C, V) : Δ
\triangleright C: knowledge base constraints
\triangleright V: knowledge base variables
$\triangleright \Delta$ set with constraints
for all $v_i \in V$ do
for all $v_j \in V v_i eq v_j$ do
$A = \emptyset;$ \triangleright set with assignments
for all $dom_k \in dom(v_i)$ do
dompair = false;
$C' \leftarrow C \cup \{v_i = dom_k\}$
for all $dom_l \in dom(v_j)$ do
$C'' \leftarrow C' \cup \{v_j \neq dom_l\}$
if $inconsistent(C'') \wedge dompair = false$ then
dompair = true;
$A \leftarrow A \cup \{v_i = dom_k \land v_j = dom_l\}$
end if
end for
end for
if $ A = dom(v_i) $ then
$\Delta \leftarrow \Delta \cup disjunctive(A)$
end if
end for
end forreturn Δ

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

To get a deep understanding of the anomalies we need to explain them to the knowledge engineer [2]. For the calculation of an explanation we use the QuickXPlain algorithm (see Algorithm 2) for each type of anomaly. We take the set of constraints (e.g. set of dead elements) and add this set to C_{KB} in the algorithm. Now we have ensured, that the constraint which describes the anomaly, can't be part of Δ in the algorithm. Next, we have to negate the constraint set which describes the anomaly. Since the negation of the anomaly can never be consistent, QuickXPlain will return the set of constraints which is responsible for the anomaly. For example, the dead domain element UniCycle = true will be negated and added to C. In that case, QuickXPlain will return the set $\{c_8\}$ as an explanation for the dead domain element.

3.3 Simulation

Due to the huge complexity of calculating all possible instances for all possible assignments (see Section 2) in constraint-based configuration systems we use Gibbs' simulation to estimate the consistency rate cr for a specific set of assignments A [11]. With this approximation, we can ...

... estimate the restriction rate (number of consistent instances compared to all instances) and evaluate the knowledge base (see Section 3.4).

... generate test cases for boundary value analysis [11].

...rank diagnoses and conflicts (assuming that a knowledge base with nearly the same restriction rate compared to the current knowledge base is preferred).

... generate reports for variety management (e.g. 'How many bikes can be used for *Competition*, *EverydayLife*, and *HillClimbing*?').

An assignment is a constraint which contains one variable a_v , one domain element a_d , and a relationship between variable and domain element a_r (see Section 2). Examples for assignments are eBike = true; and BikeType = MountainBike. Algorithm 11 is divided into three functions and shows the basic algorithm for estimating the consistency rate for a set of assignments.

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

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

Algorithm 11 GibbsSampling function GIBBS(KB, A): Δ $CC = \emptyset$ \triangleright set of consistency check results $\{0, 1\}$ mincalls = 200⊳ constant threshold = 0.01⊳ constant consistent = 0 $verify = Double.Max_Value$ while $n < mincalls \lor verify > threshold$ do $randA = A \cup GENERATERANDASSIGN(KB)$ C.addAll(randA) $\triangleright C \in KB$ if isConsistent(KB) then consistent + +CC.add(1)else CC.add(0)end if C.removeAll(randA)verify = verifyChecks(CC)n + +end while **return** consistent/n end function function GENERATERANDASSIGN(KB): A $A = \emptyset$ \triangleright A: set of assignments n = Random(C) > 0: ▷ generate n assignments for i = 0; i < n; i + + do $\triangleright \, V \in KB$ $a_v = Random(V)$ $a_r = Random(R)$ $a_d = Random(dom(a_v))$ A.add(a)end for return A end function function VERIFYCHECKS(CC): Δ $CC_1 = CC.split(0, |CC|/2)$ $CC_2 = CC.split((|CC|/2) + 1, |CC|)$ $mean1 = mean(CC_1)$ $mean2 = mean(CC_2)$ if $mean1 \ge mean2$ then **return** mean1 - mean2 else return mean2 - mean1 end if end function

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

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

Knowledge base Evaluation 3.4

As mentioned in the previous Sections, we can analyze a knowledge base in different ways and collect a lot of information about the knowledge base. Finally, we can also evaluate the knowledge base in terms of metrics. Those metrics a) help to get information about the quality of the knowledge base and b) get information about the quality of previous changes. Next, we will describe some metrics, use them to answer five questions, and measure three goals for constraint-based configuration systems (goal-question-metrics). The metrics are based on a literature review focusing on knowledge engineering. An overview of the literature review is given in [10]. In the following list we describe several metrics.

- Number of variables $|V| \in KB$.
- Average domain size domsize: $\frac{\sum_{v_i \in V} |dom(v_i)|}{|V|}$ Number of constraints $|C_{KB}| \in KB$
- Number of minimal conflicts |CS|: see Definition 3
- Minimal cardinality CS MCCS: the lowest number of constraints in a conflict set
- Number of minimal diagnoses $|\Delta|$: see Definition 5
- Minimal cardinality diagnosis $MC\Delta$: the lowest number of constraints in a diagnosis
- Number of redundancy sets |R|
- Maximal cardinality redundancy set MCR: the largest number of • constraints in a redundancy set
- dead elements DE: number of dead elements compared to the • total number of all domain elements

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

• full mandatory FM: number of full mandatory domain elements compared to the total number of all domain elements

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

- unnecessary refinement UR: whenever a variable v_i has an assignment, we can predict the assignment of variable v_j , s.t. $dom(v_i) \rightarrow dom(v_j)$
- restriction rate RR: $\frac{|C|}{|V|}$

F

- restriction rate RR_2 : $\frac{\sum_{c_i \in C} \frac{\#vars(c_i)}{\#vars(C)}|C|}{|V|}$ where $\#vars(c_i)$ is the number of variables in c_i .
- variable influence factor $VIF(v_i)$: number of constraints in which a variable v_i appears related to the number of constraints,

$$\text{e.g., } VIF(v_i) = \frac{\sum_{c_i \in C} \begin{cases} 1 & v_i \in c_i \\ 0 & else \end{cases}}{|C|}$$

- variable influence factor VIF_{all} : average influence of all variables $\sqrt{(VIF(v_i) - \frac{\sum_{v_j \in V} VIF(v_j)}{|V|})^2}$
- coverage coverage: GIBBSSAMPLING(KB, \emptyset) (see Section 3.3)

With the metrics we collected a lot of information about the knowledge base. To evaluate the knowledge base, we aggregate the metrics and use the goal-question-metrics approach [1] to quantify the quality of the knowledge base.

The aggregation of metrics will be used to answer questions relating to one or more goals. Next, we are listing the questions and the corresponding metrics for each question:

- Q1: Is the configuration knowledge base complete?: |V|, domsize, |C|, coverage, |CS|, $|\Delta|$
- Q2: Does the knowledge base contain anomalies?: $|CS|, |\Delta||R|, DE, FM, UR$
- Q3: Does the configuration knowledge base have an admissible performance?:
 - |V|, domsize, |C|, |R|DE, FM, UR
- Q4: Is the configuration knowledge base modifiable?: $MCCS, MC\Delta, MCR, DE, FM, UR, RR, RR_2,$ $VIF_{all}, Coverage$
- Q5: Is the configuration knowledge base understandable?: $MCCS, MC\Delta, MCR, DE, FM, UR, RR, RR_2,$ $VIF_{all}, coverage$

Based on the answers for these questions we can evaluate the quality of a knowledge base. The quality will be measured in terms of three goals which we will list in the following:

- G1: A configuration knowledge base must be **maintainable**, such that it is easy to change the semantics of the knowledge base in a desired manner (corresponding questions: Q2 anomalies, Q4 modifiability)
- G2: A configuration knowledge base must be **understandable**, such that the effort for a maintainability task for a knowledge engineer can be evaluated (corresponding questions: Q2 anomalies, Q5 understandability)
- G3: A configuration knowledge base must be **functional**, such that it represents a part of the real world (e.g. a bike configuration knowledge base; corresponding questions: Q1 completeness, Q2 anomalies, Q3 performance).

The results of the GQM-approach can be explained by a comparison with the measurements of previous versions of the knowledge base. The comparison can show, if maintainability, understandability, and functionality increases or decreases over time and explain the changes (based on a comparison of the answers for the questions and metrics). For a detailed description of the GQM-approach for constraint-based configuration systems we refer the reader to [10].

4 Summary

In this paper we presented approaches to improve the maintenance for constraint-based configuration systems. We described the state-of-the-art in conflict and redundancy management and introduced recommendation for the support of knowledge engineers. New anomaly detection algorithms can be used to detect well-formedness violations. Simulation techniques in the context of constraint-based configuration systems allow us to approximate metrics for the goalquestion-metrics approach. We implemented these approaches in our web-based system called 'iCone' (Intelligent environment for the development and maintenance of configuration knowledge bases) [15].⁴

While we presented novel approaches to support knowledge engineers, further research has to be done in the *verification* of the new recommendation, simulation, and metrics evaluation techniques. Furthermore, *micro tasks* can be used to collect and verify assumptions of knowledge engineers about the knowledge base. Further research should also be done in the context of *stakeholder integration*. For example, in the software engineering process it is common that several stakeholders (e.g. customers and users) can participate in the engineering process. For the integration of different stakeholders and to optimize the knowledge engineering, further research should also be done in the context of *knowledge engineering processes* and *knowledge base development*.

REFERENCES

- Victor R. Basili. Software modeling and measurement: The goal/question/metric paradigm. Technical report, University of Maryland at College Park College Park, MD, USA, College Park, MD, USA, 1992.
- [2] Alexander Felfernig, David Benavides, Jos A. Galindo, and Florian Reinfrank. Towards anomaly explanation in feature models. *Workshop on Configuration*, pages 117 – 124, 2013.
- [3] Alexander Felfernig, Paul Blazek, Florian Reinfrank, and Gerald Ninaus. Intelligent User Interfaces for Configuration Environments, volume 1, pages 89 – 106. Morgan Kaufmann, 2014.
- [4] Alexander Felfernig, Stefan Reiterer, Martin Stettinger, Florian Reinfrank, Michael Jeran, and Gerald Ninaus. Recommender systems for configuration knowledge engineering. *Workshop on Configuration*, pages 51 – 54, 2013.
- [5] Alexander Felfernig, Christoph Zehentner, and Paul Blazek. Corediag: Eliminating redundancy in constraint sets. In Martin Sachenbacher, Oskar Dressler, and Michael Hofbaur, editors, DX 2011. 22nd International Workshop on Principles of Diagnosis, pages 219 – 224, Murnau, GER, 2010.
- [6] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems: An Introduction*, volume 1. University Press, Cambridge, 2010.
- [7] Ulrich Junker. Quickxplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artifical intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004.
- [8] Michael Pazzani and Daniel Billsus. Content-based recommendation systems. In Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl, editors, *The Adaptive Web*, volume 4321 of *Lecture Notes in Computer Science*, pages 325–341. Springer Berlin / Heidelberg, 2007.
- [9] Cédric Piette. Let the solver deal with redundancy. In Proceedings of the 2008 20th IEEE International Conference on Tools with Artificial Intelligence - Volume 01, pages 67–73, Washington, DC, USA, 2008. IEEE Computer Society.
- [10] Florian Reinfrank, Gerald Ninaus, Bernhard Peischl, and Franz Wotawa. A goal-question-metrics model for configuration knowledge bases. *Configuration Workshop*, 2015.
- [11] Florian Reinfrank, Gerald Ninaus, Franz Wotawa, and Alexander Felfernig. Maintaining constraint-based configuration systems: Challenges ahead. *Configuration Workshop*, 2015.
- [12] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
- [13] Stuart Russell and Peter Norvig. Artificial Intelligence. A modern approach, volume 2. Prentice Hall, New Jersey, 2003.
- [14] Edward Tsang. Foundations of Constraint Satisfaction. Academic Press, 1993.
- [15] Franz Wotawa, Florian Reinfrank, Gerald Ninaus, and Alexander Felfernig. icone: intelligent environment for the development and maintenance of configuration knowledge bases. *IJCAI 2015 Joint Workshop* on Constraints and Preferences for Configuration and Recommendation, 2015.
- [16] Franz Wotawa, Martin Stettinger, Florian Reinfrank, Gerald Ninaus, and Alexander Felfernig. Conflict management for constraint-based recommendation. *IJCAI 2015 Joint Workshop on Constraints and Pref*erences for Configuration and Recommendation, 2015.

Juha Tiihonen, Andreas Falkner and Tomas Axling, Editors Proceedings of the 17th International Configuration Workshop September 10-11, 2015, Vienna, Austria

⁴ http://ase-projects-studies.ist.tugraz.at: 8080/iCone/