Maintaining Constraint-based Configuration Systems: Challenges ahead

Florian Reinfrank, Gerald Ninaus, Franz Wotawa, Alexander Felfernig
Institute for Software Technology
Graz University of Technology
8020 Graz, Austria
firstname.lastname@ist.tugraz.at

Abstract. Constraint-based configuration systems like knowledge-based recommendation and configuration are used in many different product areas such as cars, bikes, mobile phones, and computers. The development and maintenance of such systems is a time-consuming and error prone task because the content of such systems and the responsible knowledge engineers are changing over time.

Much research has been done to support knowledge engineers in their maintenance task. In this paper we give a short overview of previous research in the context of intelligent techniques to support the maintenance task and give an overview of future research aspects in this area. This paper focuses on intelligent simulation techniques for generating metrics, predicting boundary values for automated test case generation, assignment-based (instead of constraint-based) anomaly management, and processes for the development of constraint-based configuration systems.

1 Introduction

The number of e-commerce web sites and the quantity of offered products and services is increasing enormously [2]. This triggered the demand of intelligent techniques that improve the accessibility of complex item assortments for users. Such techniques can be divided into configuration-based systems and recommendation systems. When the e-commerce system has highly configurable products (e.g. cars), configuration-based systems can help users to configure the product based on their needs. If, on the other hand, the e-commerce system contains many different products, intelligent recommendation techniques can help to find the product, which fits best to the user's needs [16]. We can differentiate between collaborative systems (e.g., www.amazon.com [16]), content-based systems (e.g., www.youtube.com [16]), critiquing-based systems (e.g., www.movielens.org [4]), and constraint-based recommendation systems (e.g., www.myproductadvisor.com [6]). The favored type of recommendation systems depends on the domain in which the recommendation system will be used. For example, in highly structured domains where almost all information about a product is available in a structured form, critiquing and constraint-based recommendation systems are often the most valuable recommendation approach.

Such systems are used, for example in the notebook domain. Such product domains - like the notebook domain - change over time because new product characteristics can fit to new customer needs. For example, ten years ago the number of cpu cores for notebooks was not a configurable variable. Nowadays, users can choose between one, two, or four cpu cores. This example shows that constraint-based recommendation systems have to be updated over time. While adding a variable to the product might be easy to handle, adding and editing constraints can be a time consuming and error prone task. This problem occurs in complex constraint-based recommendation systems with many existing constraints.

A lot of research has been done in the last years to tackle this challenge. For example, recommendation techniques can help to support knowledge engineers in their maintenance tasks, via reducing the sets of constraints so that the engineer can focus on the relevant constraints. Other examples for the support of the maintenance tasks are anomaly detection, dependency detection, and metrics measurement. An example application for the maintenance of constraint-based configuration systems is iCone (Intelligent Environment for the Development and Maintenance of configuration knowledge bases) [21, 26].[1]

Based on intelligent techniques to support knowledge engineers in their maintenance tasks, this paper focuses on further aspects in the maintenance of constraint-based configuration systems and picks up four research aspects (see below) for improving existing development and maintenance environments for constraint-based configuration systems like knowledge-based configuration and recommendation systems.

The paper is organized as follows: Section 2 (preliminaries) gives an overview of constraint-based configuration systems and a running example. Section 3 contains four aspects of the context of constraint-based configuration system development and maintenance. Section 3.1 is dealing with simulation techniques for constraint-based configuration systems. Section 3.2 shows principles of test case generation based on software engineering for constraint-based systems. An introduction for assignment-based anomaly management is given in Section 3.3. Section 3.4 goes beyond maintaining constraint-based configuration systems and takes a look into development pro-

[1] `http://ase-projects-studies.ist.tugraz.at:8080/iCone`

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

cesses for configuration systems. Section 4 summarizes this paper.

2 Preliminaries

In this Section we will describe constraint-based configuration systems, the terms assignment, consistency, and redundancy, and introduce a short knowledge-based recommendation system for notebooks as an example for a constraint-based configuration system.

We use the terminology of constraint satisfaction problems (CSP) [25] to represent configuration systems. Constraint-based configuration systems are defined as a triple $KB = \{V, D, F\}$. V is a set of product and customer variables. All variables have a selection strategy v_{sel} which describes, if a variable can have more than one value. $v_{sel} = singleselect$ shows that the variable v can have zero or one assignment, e.g. each product has one *price*, e.g. $price = 399$. If a variable can have more than one value, we differ between $multipleAND$ and $multipleOR$. $v_{sel} = multipleAND$ points out that a variable can have more than one assignment. For example, a notebook can have two wireless connections like *bluetooth* AND $WLAN$, s.t. $wireless\_connection_{sel} = multipleAND$. On the other hand, a customer wants to have a notebook with two OR four *cpu_cores*. We denote such a selection strategy as $multipleOR$, s.t. $cpu\_cores_{sel} = multipleOR$.

Each variable $v_i \in V$ has a domain $dom(v_i) \in D$ that contains the set of all possible values (not only the assigned values). Each variable can have *zero to n* finite assignments. Products F_P, customer requirements F_R, and constraints which are defining the relationship between product variables and customer variables F_C are in the filter set F.

Customer requirements represent the preferences of customers in the recommendation / configuration process. The set of customer preferences is denoted as F_R. For example, a customer can have the preference that a notebook should be cheaper than 599 EUR, s.t. $\{price < 599\} \in F_R$. Furthermore, customers can be asked for their usage scenarios, which might can be $multimedia, office, gaming$. If a user has more than one usage scenario, we duplicate the variable *usage_scenario* for this user, s.t. $usage\_scenario1 = multimedia \wedge usage\_scenario2 = office$.

Constraints which can also be denoted as filters in F_C define the relationship between customer preferences and product variables and are defined in the set $F_C \in F$. For example, the relationship between the customer's *usage_scenario* and the product attributes is $f_1 := usage\_scenario = gaming \rightarrow cpu\_cores > 2$. Additionally, constraint-based recommendation systems have a set of products. This set is denoted as $F_P \in F$ and contains one disjunctive query with all products, s.t. $F_P = \{product_0 \vee product_1 \vee ... \vee product_n\}$ and each product is a conjunctive query of the product variables, s.t. $product_0 = \{price = 399 \wedge cpu\_cores = 2\}$. The aggregation of customer requirements, constraints, and products represent the filters, s.t. $F_R \cup F_C \cup F_P = F$.

Each filter can be divided into **assignments**. An assignment consists of a variable v, a relationship, and a value d which is an element of the domain $dom(v)$. The different types of relationships depend on the values in the corresponding domain. If, for example, the domain consists only of numbers, we can say that the types of relationships are $<, \leq, =, \neq, \geq, >$ whereas for domains with strings we reduce the different types of relations to $=, \neq$.

To consider the selection strategy of variables within the filters, we have to duplicate the variables. For example, if a customer wants to have a notebook for *gaming* and $office$, we replace the variable $usage\_scenario \in V$ with $usage\_scenario1 \in V$ and $usage\_scenario2 \in V$ with the same domain in the knowledge base KB. We also have to extend the affected filters in F, such that we have to replace the affected assignments in the example constraint $usage\_scenario = gaming \rightarrow cpu\_cores > 2 \in F_C$ with the assignments $(usage\_scenario1 = Gaming \vee usage\_scenario2 = Gaming) \rightarrow cpu\_cores > 2 \in F_C$.

To check if at least one product fits to the customer's preferences, we do consistency checks, s.t. $V \cup D \cup F \neq \emptyset$. If at least one product in the constraint-based configuration system is presented to the customer, we can say, that the knowledge base is **consistent**. Otherwise, the knowledge base contains inconsistencies. For dealing with inconsistencies, we refer the reader to [3, 5, 10, 11, 12, 15].

If the knowledge base is consistent, we can further evaluate whether the knowledge base contains **redundancies**. A redundancy is given, if the removal of a constraint from F_C leads to the same semantics [15, 20].

In the following we describe a notebook domain. The simplified domain is represented as a knowledge-based recommendation system.

$$V = \{price, cpu\_cores, usage\_scenario\}$$
$$price_{sel} = singleselect$$
$$cpu\_coressel = singleselect$$
$$usage\_scenariossel = multipleAND$$

$$D = \{$$
$$dom(price) = \{399, 599, 799, 999\},$$
$$dom(cpu\_cores) = \{2, 4\},$$
$$dom(usage\_scenario) = \{office, multimedia, gaming\}$$
$$\}$$

$$F_C = \{$$
$$f_1 := usage\_scenario = office \rightarrow (price < 599 \wedge cpu\_cores = 2)$$
$$f_2 := usage\_scenario = multimedia \rightarrow ((price < 999 \wedge cpu\_cores = 4) \vee price < 799)$$
$$f_3 := usage\_scenario = gaming \rightarrow cpu\_cores = 4$$
$$\}$$

$$F_R = \emptyset$$

$$F_P = \{$$
$$(price = 399 \wedge cpu\_cores = 2) \vee \quad (p_0)$$
$$(price = 599 \wedge cpu\_cores = 4) \vee \quad (p_1)$$
$$(price = 799 \wedge cpu\_cores = 2) \vee \quad (p_2)$$
$$(price = 999 \wedge cpu\_cores = 4) \quad (p_3)$$
$$\}$$

$$F = F_C \cup F_R \cup F_P$$

$$KB = V \cup D \cup F$$

3 Challenges in the Development and Maintenance of Constraint-based Configuration Systems

In the following, we describe basic approaches to increase the maintainability, understandability, and functionality of constraint-based configuration systems. Therefore we use simulation techniques (Section 3.1), automated test case generation (Section 3.2), assignment-based anomaly management (Section 3.3), and the consideration of the development process (Section 3.4).

3.1 Simulation

In the context of constraint-based configuration systems we use simulation to approximate the number of consistent constraint sets compared to the number of all possible constraint sets. This technique can be used to calculate metrics - like the number of valid configurations - or to approximate the dependency between variables [21]. On the one hand we loose minimal accuracy when calculating the possible number of consistent constraint sets whereas, on the other hand, it is possible to approximate metrics which can not be calculated in an efficient manner. In the following, we describe the basic functionality of simulation for constraint-based configuration systems and give an example simulation in Figure 1.

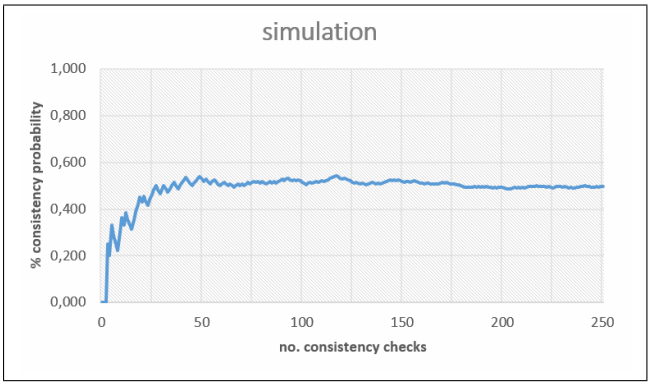

Figure 1. Example simulation for approximated consistency. We assume that a high number of consistency checks leads to a representative sample of the configuration knowledge base (Law of large numbers). In this example the average number of consistent configurations is approx. 50%.

Due to the huge complexity for calculating all possible instances for all possible assignments in constraint-based configuration systems, we use Gibbs' simulation to estimate the consistency rate *coverage* for a specific set of assignments A [22]. An assignment is a filter 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). Algorithm 1 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 filters $C \in KB$ (see Section 2). The set CC (checks) contains all results from consistency checks.

Algorithm 1 GibbsSampling

```
function GIBBS(KB, A): Δ
    CC = ∅                          ▷ set of consistency check results {0,1}
    mincalls = 200                                              ▷ constant
    threshold = 0.01                                            ▷ constant
    consistent = 0
    verify = Double.Max_Value
    while n < mincalls ∨ verify > threshold do
        randA = A ∪ GENERATERANDASSIGN(KB)
        F.addAll(randA)                                         ▷ F ∈ KB
        if isConsistent(KB) then
            consistent++
            CC.add(1)
        else
            CC.add(0)
        end if
        F.removeAll(randA)
        verify = VERIFYCHECKS(CC)
        n++
    end while
    return consistent/n
end function
function GENERATERANDASSIGN(KB):A
    A = ∅                                   ▷ A: set of assignments
    n = Random(F ∈ KB):                     ▷ generate n assignments
    for i = 0; i < n; i++ do
        a_v = Random(V ∈ KB)                                    ▷ V ∈ KB
        a_r = Random(Rel)
        a_d = Random(dom(a_v) ∈ D ∈ KB)
        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 ≥ mean2 then
        return mean1 − mean2
    else
        return mean2 − mean1
    end if
end function
```

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 the programming variable *consistent*. *threshold* is a constant and required to test if the current set of consistency checks has a high accuracy. The variable $verify$ contains the result of the last verification returned by the function $VERIFYCHECKS$. If the variable $verify$ is greater than the *threshold*, we can not guarantee that the current result is accurate. In that case we have to execute the loop again. In the while-loop we first have to generate a new set of random assignments. Since assignments are also special types of constraints, we add the set $randA$ to the set $F_C \in KB$ and do a consistency check. If KB with the randomly generated assignments is consistent, we add 1 to the set CC and incre-

ment 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. $Random(F)$ returns the number of assignments which have to be generated randomly. This number depends on the number of filters in the knowledge base, the number of available variables and domain elements, since in small knowledge bases it can happen that we can not generate more than *mincalls* assignments. $Random(V)$ takes a variable from the knowledge base. If the variable is already part of another assignment, the variable won't be used again except the selection strategy v_{sel} is either $multipleAND$ or $multipleOR$. $Random(R)$ selects a relation between the variable and the domain elements. In our case, variables can have textual domain elements (e.g. the brand of a notebook) or numeric domain elements (e.g. the price of a notebook). While the set of relations for textual domain elements is $Rel = \{=, \neq\}$, the set is extended to $Rel = \{=, \neq, <, \leq, >, \geq\}$ for numerical domain elements (see Section 2). Finally, $Random(dom(a_v))$ selects a domain element from $dom(a_v)$ randomly.

The function $verifyChecks(CC)$ tests if the numbers 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 $mean(CC_1)$ and $mean(CC_2)$ are close to each other. If they have nearly the same values, $(\sqrt{(mean(CC_1) - mean(CC_2))^2} \leq threshold)$, we can say that the consistent checks are normally distributed and return the difference between $mean(CC_1)$ and $mean(CC_2)$.

In our iCone implementation we use the simulation technique in three different ways. First, we evaluate the *coverage* metric which defines the number of consistent configurations compared to the number of all possible configurations [22]. Second, we use this technique to generate random assignments for test cases (see Section 3.2). Finally, we use this technique to approximate the consistency rate *coverage* for at least two variables and their domain elements. Figure 2 shows the probability that the combination of two assignments is consistent. For example, approximately 100% of the notebook configurations are consistent, if the $usage\_scenario = multimedia \wedge cpu\_cores = 2$.

3.2 Test Case generation

In this Section we want to describe a basic approach to generate test cases for constraint-based configuration systems.

In software engineering, boundary value analysis are *those situations directly on, above, and beneath the edges of input equivalence classes* [19]. To use this type of software testing in the context of configuration systems, we can say that the *edges* are within variable assignments. For example, if $price = 399$ is consistent, $price = 599$ is consistent too, and $price = 799$ is inconsistent, the boundary would be between the domain elements 599 and 799. In Figure 2 we can see that, under circumstances, some combinations are inconsistent (e.g.

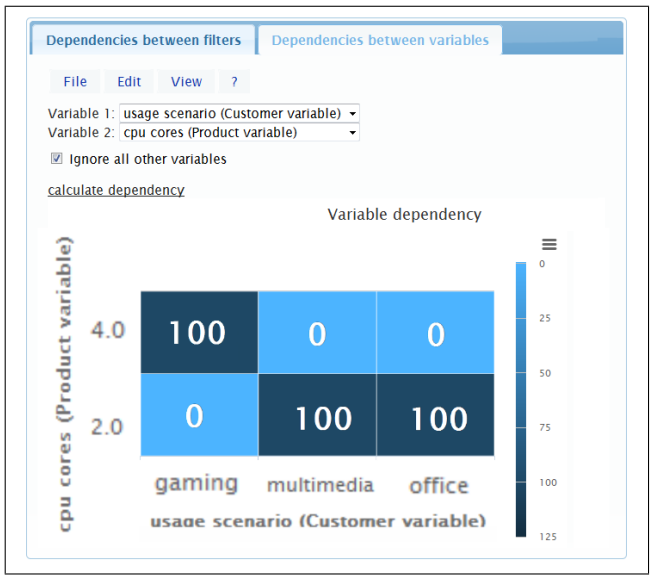

Figure 2. Example simulation for approximated consistency

$usage\_scenario = multimedia \wedge cpu\_cores = 2$) and some are consistent (e.g. $usage\_scenario = gaming \wedge cpu\_cores = 2 = inconsistent$). We can use the simulation technology (see Section 3.1) to generate various sets of filter constraints to get some boundaries. Table 1 shows a list of randomly generated test cases. Note that the number of assignments in the test case can be different (see Algorithm 1).

tc	$filterconstraint$	$coverage$
t_0	$cpu\_cores = 2 \wedge$ $usage\_scenario = office$	0.50
t_1	$cpu\_cores = 2 \wedge$ $usage\_scenario = multimedia$	0.50
t_2	$price = 799 \wedge$ $usage\_scenario = gaming$	0.00
t_3	$price = 599 \wedge$ $usage\_scenario = gaming$	0.50
t_4	$cpu\_cores = 4 \wedge$ $usage\_scenario = multimedia$	0.50
t_5	$cpu\_cores = 4$	~ 0.54

Table 1. An example for randomly generated test cases.

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

After the randomly detected boundaries via simulation we have to evaluate the boundary. Such evaluations have to be done by stakeholders of the knowledge base and can be done via micro tasks [7]. In this context, several stakeholders can be asked if the results of randomly generated test cases are valid or not. Such answers can be collected within a case base. Table 2 gives an example case base.

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

stakeholder	*testcase*	*correct?*
s_0	t_2	*yes*
s_1	t_2	*yes*
s_2	t_2	*yes*
s_3	t_2	*yes*
s_4	t_2	*yes*
s_5	t_2	*no*
s_6	t_2	*yes*
s_7	t_2	*yes*
s_0	t_3	*no*
s_1	t_3	*no*
s_2	t_3	*yes*
s_3	t_3	*yes*
s_4	t_3	*no*
s_5	t_3	*no*
s_6	t_3	*no*
s_7	t_3	*yes*

Table 2. An example case base for evaluating randomly generated test cases.

ers such as mails, forum, and content-based recommendation [16].

Finally, a result of the discussion leads to a consistent knowledge base (filter constraints $f \in F_C$ have to be updated or removed) which represents the real product domain. If the knowledge base has to be maintained, intelligent techniques like the detection of minimal conflicts and diagnoses [21] help to detect the causes for the difference between the knowledge base and the real world.

3.3 Assignment-based anomaly management

The anomaly management research describes different approaches to detect and explain anomalies [21, 26]. For example, QuickXplain can detect conflicts [17], FastDiag finds minimal diagnoses for these conflicts [13], Sequential [20] and CoreDiag [15] can remove maximal sets of filter constraints without changing the semantics of the knowledge base (redundancy detection). Well-formedness violations can detect domain elements which can never be selected (*deadelements*) or have to be selected ($fullmandatories$) or can only exit if specific domain elements of other variables are selected as well ($unnecessaryrefinements$) [21].

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

Alternatively, we can check the assignments within a filter instead of the filter itself for anomalies. Algorithm 2 gives an example for an assignment-based algorithm. This algorithm extends the *Sequential* algorithm introduced by Piette [20]. First of all, we have to generate the negation of all filter constraints in the knowledge base. We denote the negation $\bar{F_C}$ and define a disjunctive query of the original knowledge base $\neg f_1 \vee \ /f_2 \vee \ /f_3$. If the negation of the knowledge base in combination with the original knowledge base is inconsistent, s.t. $F_C \cup \bar{F_C}$ is inconsistent, the knowledge base has not changed its semantics. If a filter will be removed from the knowledge base (but not from the negation of the knowledge base) and the combination is still inconsistent, we can say that the knowledge base has kept its semantics and the removed filter constraint is redundant.

Algorithm 2 AssignmentSequential

```
function AssignmentSequential(KB): R
                                   ▷ KB: knowledge base
  F̄_C = ¬f_1 ∨ ¬f_2 ∨ ¬f_3
  R = ∅
  for all f ∈ F_C do
    for all a ∈ A(f) do
      A.remove(a)
      if (F_C ∪ F̄_C) isinconsistent then
        R.add(a)
      else
        A.add(a)
      end if
    end for
  end for
  return R
end function
```

While the Sequential algorithm removes filter constraint by filter constraint from F_C, we divide the filter constraint into its assignments and remove assignment by assignment. Therefore, we introduce the set $A(f)$ which describes the set of assignments of filter constraint $f \in F_C$. When we remove an assignment from $A(f)$, we next have to consider the relations between the assignments. Figure 3 shows the graphical representation of all filter constraints and their assignments in our example knowledge base in a conjunctive order. When we remove an assignment a from $A(f)$ we will further replace the upper relation. For example, the removal of the assignment $usage\_scenario = office$ of filter f_1 replaces the upper implication $\rightarrow$ with the top node of those elements which will not be connected to the conjunctive constraint. In our case, this is relation '$\wedge$'.

Algorithm 2 introduces an approach to detect redundant assignments within a knowledge base. The approach is straight forward: First, we have to generate the negation of $\bar{F_C}$. Then we select filter by filter. For each filter we remove assignment by assignment a. Finally, we check if the knowledge base with the changed filter f is still inconsistent with $\bar{F_C}$. If it is inconsistent, we can say that the removed assignment a is redundant.

Figure 4 shows the redundant assignments of our example knowledge base. In the first row we see the original filters and the result for the $usage\_scenario$ variable (green box). Then we remove assignment by assignment and see the result of the filter constraints in the column *result*. The yellow boxes suggest that the adapted filter constraints lead to the same semantics as the original knowledge base. We can remove $cpu\_cores = 2$ from filter constraint f_1 and the assignments $price < 999$ and $cpu\_cores = 4$ from filter constraint f_2 without changing the semantics of the knowledge base.

Similar adaptations can also be done e.g., for QuickXPlain [17], FastDiag [14], and CoreDiag [15]. While these algorithms use a divide and conquer approach based on filters, future research can also consider assignments instead of filters to calculate the anomalies.

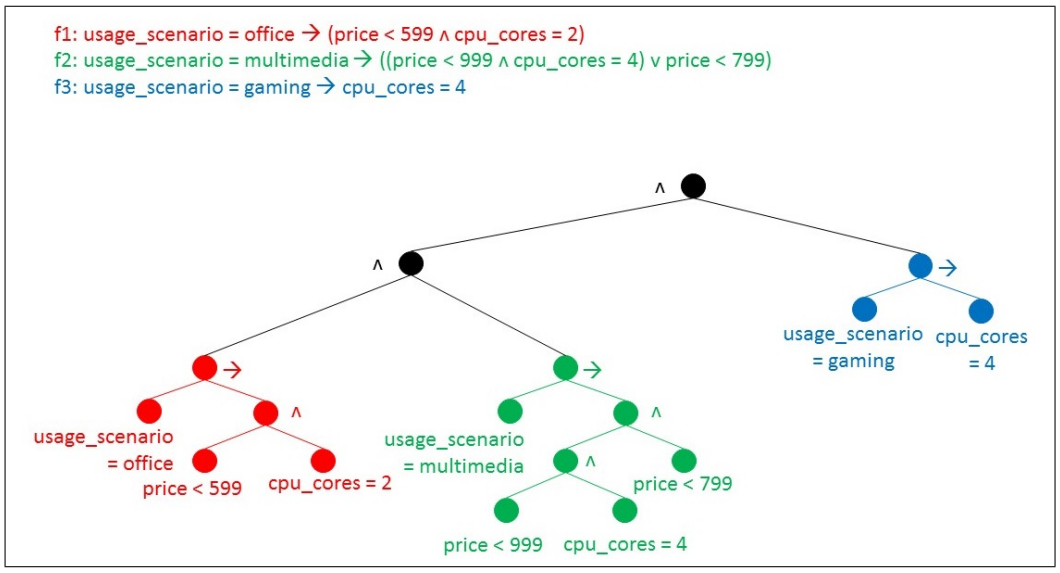

Figure 3. Conjunctive query of all filters $f \in F_C$ in our example knowledge base. In Algorithm 2 we remove assignment by assignment from the knowledge base and check the consistency instead of the whole filter (f_1, f_2, f_3).

f1	f2	f3	result		redundant
usage_scenario = office → (price < 599 ∧ cpu_cores = 2)	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) v price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	p0 p0, p1 p1, p3	
price < 599 ∧ cpu_cores = 2	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) v price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	p0 ∅ ∅	
usage_scenario = office → cpu_cores = 2	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) v price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	p0,p2 p0, p1 p1, p3	
usage_scenario = office → price < 599	usage_scenario = multimedia → ((price < 999 ∧ cpu_cores = 4) v price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	p0 p0, p1 p1, p3	cpu_cores = 2
usage_scenario = office → price < 599	((price < 999 ∧ cpu_cores = 4) v price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	p0 p0, p1 p1	
usage_scenario = office → price < 599	usage_scenario = multimedia → (cup_cores = 4 v price < 799)	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	p0 p0, p1 p1, p3	price < 999
usage_scenario = office → price < 599	usage_scenario = multimedia → price < 799	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	p0 p0, p1 p1, p3	cpu_cores = 4
usage_scenario = office → price < 599	usage_scenario = multimedia	usage_scenario = gaming → cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	∅ p0, p1, p2, p3 ∅	
usage_scenario = office → price < 599	usage_scenario = multimedia → price < 799	cpu_cores = 4	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	∅ p1 p1, p3	
usage_scenario = office → price < 599	usage_scenario = multimedia → price < 799	usage_scenario = gaming	usage_scenario = office usage_scenario = multimedia usage_scenario = gaming	∅ ∅ p0, p1, p2, p3	

Figure 4. Results for consistency checks. The columns f_1, f_2, and f_3 show the filter constraints when one assignment will be removed. The first row shows the results (green background) of the original filter constraints. The yellow background suggests, that the removal of the assignments leads to the same results.

3.4 Constraint-based configuration system development

A lot of research has been done in the maintenance of constraint-based systems. For example, we can evaluate the quality of knowledge bases [22] and check if the knowledge base has anomalies [5, 21]. Therefore, we can evaluate if we are doing the knowledge base maintenance efficiently.

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

Development processes for constraint-based configuration systems

An overview of knowledge base engineering processes is given in [9, 24].

Common-KADS focuses on different models (organization, task, agent, communication, and expertise) of the knowledge base. For example, the expertise model tries to describe knowledge from a static, functional, and a dynamic view. While this system tries to consider all stakeholders, it does not prioritize the knowledge and does not try to solve conflicts in the knowledge before it will be transferred into a constraint-based configuration system [23].

The **MIKE** engineering process can be seen as an iterative process and is divided into the activities *elicitation*, *interpretation*, *formalization / operationalization*, *design*, and *implementation*. The entire development process, i.e. the sequence of knowledge acquisition, design, and implementation, is performed in a cycle inspired by a spiral model as process model. Every cycle produces a prototype as output which can be evaluated by tests in the real target environment. The evaluation of each activity will be done by domain experts. While the result of the implementation activity can be evaluated by domain experts, a deep understanding of modelling techniques is required to evaluate the results of elicitation, interpretation, and formalization activities [1].

Protege-II is used to model method and domain ontologies. A method ontology defines the concepts and relationships that are used by a problem solving method for providing its functionality. Domain ontologies define a shared conceptualization of a domain. Both ontologies can be reused in other domains which may reduce the effort to build-up a new knowledge base with similar elements [18].

Development in the Software Engineering Discipline

Compared to development processes for constraint-based configuration systems we give an overview of actual trends in the engineering of such systems and create a link to the currently existing development processes for constraint-based configuration systems.

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

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

4 Conclusion

In this paper we gave an overview of future research in the context of developing and maintaining constraint-based configuration systems. Such systems can be constraint-based configuration, knowledge-based recommendation systems, or feature models. We introduced a simulation technique in the context of constraint-based configuration systems, show some hints for automatic test case generation and gave an overview of assignment-based anomaly detection instead of constraint-based conflicts, redundancies, and well-formedness detection. Finally, we showed how requirements engineering and design patterns can be used for knowledge base engineering processes.

Acknowledgements

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).

REFERENCES

[1] J. Angele, D. Fensel, D. Landes, and R. Studer. Developing knowledge-based systems with mike. *Automated Software Engineering*, 5(4):389–418, 1998.

[2] Ivan Arribas, Francisco Perez, and Emili Tortosa-Ausina. Measuring international economic integration: Theory and evidence of globalization. *World Development*, 37(1):127 – 145, 2009.

[3] David Benavides, Alexander Felfernig, Jos A. Galindo, and Florian Reinfrank. Automated analysis in feature modelling and product configuration. *ICSR*, pages 160 – 175, 2013.

[4] Li Chen and Pearl Pu. Evaluating critiquing-based recommender agents. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, AAAI'06, pages 157–162. AAAI Press, 2006.

[5] Alexander Felfernig, David Benavides, Jos A. Galindo, and Florian Reinfrank. Towards anomaly explanation in feature models. *Workshop on Configuration*, pages 117 – 124, 2013.

[6] Alexander Felfernig and Robin Burke. Constraint-based recommender systems: technologies and research issues. In *Proceedings of the 10th international conference on Electronic commerce*, ICEC '08, pages 3:1–3:10, New York, NY, USA, 2008. ACM.

[7] Alexander Felfernig, Sarah Haas, Gerald Ninaus, Michael Schwarz, Thomas Ulz, and Martin Stettinger. Recturk: Constraint-based recommendation based on human computation. *CrowdRec*, June 2014.

[8] Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen, editors. *Knowledge-based configuration. From research to business cases*, volume 1. Morgan Kaufmann, 2014.

[9] Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen, editors. *Knowledge Engineering for configuration systems*, pages 139 – 155. Volume 1 of Felfernig et al. [8], 2014.

[10] Alexander Felfernig, Florian Reinfrank, and Gerald Ninaus. Resolving anomalies in configuration knowledge bases. *IS-MIS*, 1(1):1 – 10, 2012.

[11] Alexander Felfernig, Florian Reinfrank, Gerald Ninaus, and Paul Blazek. *Conflict Detection and Diagnosis Techniques for Anomaly Management*, pages 73 – 87. Volume 1 of Felfernig et al. [8], 2014.

[12] Alexander Felfernig, Florian Reinfrank, Gerald Ninaus, and Paul Blazek. *Redundancy Detection in Configuration Knowledge*, pages 157 – 166. Volume 1 of Felfernig et al. [8], 2014.

[13] Alexander Felfernig and Monika Schubert. Personalized diagnoses for inconsistent user requirements. *AI EDAM*, 25(2):175–183, 2011.

[14] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012.

[15] 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.

[16] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems: An Introduction*, volume 1. University Press, Cambridge, 2010.

[17] 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.

[18] Mark A Musen, Henrik Eriksson, John H Gennari, and Samson W and Tu. Protg-ii: A suite of tools for development of intelligent systems from reusable components. *Proc Annu Symp Comput Appl Med Care*, 1994.

[19] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The art of software testing*. John Wiley & Sons, 3 edition, 2012.

[20] 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.

[21] Florian Reinfrank, Gerald Ninaus, and Alexander Felfernig. Intelligent techniques for the maintenance of constraint-based systems. *Configuration Workshop*, 2015.

[22] Florian Reinfrank, Gerald Ninaus, Bernhard Peischl, and Franz Wotawa. A goal-question-metrics model for configuration knowledge bases. *Configuration Workshop*, 2015.

[23] G. Schreiber, B. Wielinga, R. de Hoog, H. Akkermans, and W. Van de Velde. Commonkads: a comprehensive methodology for kbs development. *IEEE Expert*, 9(6):28–37, Dec 1994.

[24] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data & Knowlege Engineering*, 25:161 – 197, 1998.

[25] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[26] 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.