Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2016), *30*, 313–328.

doi:10.1017/S0890060415000487

Formulating constraint satisfaction problems for the inspection of configuration rules

ANNA TIDSTAM,[1] JOHAN MALMQVIST,[1] ALEXEY VORONOV,[2] KNUT ÅKESSON,[2] AND MARTIN FABIAN[2]

[1]Department of Product and Production Development, Chalmers University of Technology, Gothenburg, Sweden
[2]Department of Signals and Systems, Chalmers University of Technology, Gothenburg, Sweden

(Received July 14, 2014; Accepted April 6, 2015)

Abstract

Product configuration is when an artifact from a product family is assembled from a set of predefined components that can only be combined in certain ways. These ways are defined by configuration rules. The product developers inspect the configuration rules when they develop new configuration rules or modify the configuration rules set. The inspection of configuration rules is thereby an important activity to avoid errors in the configuration rules set. Several formulations of constraint satisfaction problems (CSPs) are proposed that facilitate the inspection of configuration rules in propositional logic (IF-THEN, AND, NOT, OR, etc.). Many of the configuration rules are so called *production rules*; that is, a configuration rule is an IF-THEN expression that fires when the IF condition is met. Several configuration rules build chains that fire during the product configuration. It is therefore important not only to inspect single configuration rules but also to analyze the effect of multiple configuration rules. Formulating the tasks as variations of the CSP can support the inspection activity. More specifically, we address the reformulation of configuration rules, testing of feature variant combinations, and counting of item quantities from an item set. The suggested CSPs are tested on industrial vehicle configuration rules for computational performance. The results show that the time for achieving results from the solving of the CSP is within seconds. Our future work will be to implement the various CSPs into a demonstrator that could be tested by product developers.

Keywords: Configuration Rules; Constraint Satisfaction Problem; Product Configuration; Propositional Logic

1. INTRODUCTION

Product configuration is the activity to assemble one artifact from a set of predefined components when taking into account a set of restrictions on how the components can be combined (Mittal & Falkenhainer, 1990). The artifact is then called a product variant of the product family (Soininen et al., 1998). There are several methods for how to *represent* the product variants, as well as there are several methods for how to *reason* if a specific product variant belongs to a product family. It is originally the representation that is presented visually during the inspection activity. The representation methods thereby outline the possibilities for how the inspection of configuration rules could take place. Furthermore, this paper introduces reasoning methods as a complement to the representation method in order to facilitate the inspection. To describe this research work and its limitations, it is therefore important to distinguish categories of representation and reasoning methods. The categorization of reasoning methods in this paper is *rule-based, model-based*, and *case-based* methods (Junker, 2006). Included in the description of the categories is an identification of *inferable, inheritable*, and *relational* representations (Chakraborty, 2010):

- **Rule-based** methods contain formulas that have either true or false values. Rule-based representations are used for variant-rich products that are mass customized, for example, vehicle configurations. The rule-based representation has the disadvantage that it is difficult to manually analyze the combined effect of several rules. The rules may act as a chain of rules that allows or forbids product variants unexpectedly. Chains of rules can be detected by applying reasoning methods, and the chain itself is one example of a new rule that was *inferable* from the existing rules set. A rule-based reasoning method executes actions based on conditions. The ordering of the rules decides when actions are taken, which could determine the solu-

Reprint requests to: Anna Tidstam, Hörsalsvägen 7A, Göteborg 412 58, Sweden. E-mail: tidstam@gmail.com

tion. An application of rule-based reasoning methods is medical diagnosis (Shortliffe, 1976).

- **Model-based** methods are based on a structure, for example, a tree structure. Model-based representations could contain *inheritable* knowledge through links between objects, for example, *is-a, instance-of*, and *part-of* links. A feature model (Batory et al., 2006) is an example of the model-based representation, where the links between objects in a tree structure represent configuration rules. A model-based reasoning method searches/builds a structure, for example, a decision tree.
- **Case-based** methods are based on the creation of databases, for example, *relational* databases. Each product variant is stored as a string of variable values in the database. If a product variant is not found in the database, it has to be manually verified by product developers, which causes delays in the order-to-delivery process.

This paper studies the inspection of rule-based representation in propositional logic (AND, NOT, OR, IF-THEN, etc.). The system architecture of a rule-based system is shown in Figure 1. The inference engine in the figure has a reasoning method, and this paper uses a reasoning method for solving variations of constraint satisfaction problem (CSP; Tsang, 1993). The choice of rule-based representation gives, as stated above, the presence of inferable rules, which the reasoning methods can help in analyzing. The inferable rules are stored in the working memory as *facts* during the execution of the inference engine. The facts and configuration rules can then be delivered from the inference engine to the user through a user interface. The user interface presents the configuration rules using a certain *visualization method* (graphs, tables, lists, etc.).

Each time a product variant is developed or updated, the configuration rules set needs to be modified. A facilitating approach is to introduce *features* (Krebs et al., 2004). The features describe product functionality and are thus independent of specific artifacts, here called *items*. An industrial example of rule-based representation including features is described as follows:

1. **Feature variants** are product features offered in variations to the customers. Groups of feature variants describing a similar product feature are called *feature families*. Examples of feature variants for five feature families are the following:

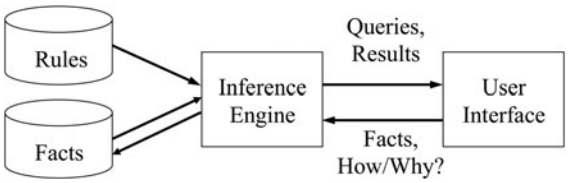

Fig. 1. System architecture for a rule-based system. Adapted from *Cloud Billing Service*, by H.-D. Wehle, 2011. http://www.ibm.com/developerworks/cloud/library/cl-devcloudmodule/. Copyright 2011 by IBM Corp. Adapted with permission.

   - Manual or automatic gearbox feature variants: {*GMAN*, *GAUT*}
   - Engine type feature variants: {*ENG1*, *ENG2*, *ENG3*}
   - Engine options feature variants: {*EVO*, *HPLOW*, *HPMED*, *HPHIG*}
   - Engine liters feature variants: {*1.6L*, *2.0L*, *2.2L*, *2.5L*, *3.0L*}
   - Steering feature variants: {*L*, *R*}
2. **Items** are needed for manufacturing the product variant (e.g., product components, CAD files, software, documents etc.). In this industrial example, there are two physical items:
   - Gearbox items: {*ITM001*, *ITM002*}
3. **Configuration rules** are logical expressions that either allow or forbid combinations of feature variants and items. Configuration rules with only feature variants are called *feature variant combination rules*. An example of a feature variant combination rule is the following:
   - IF (*GMAN*) THEN ((*R* AND (*ENG1* AND (NOT (*EVO*) OR *ENG1* AND *EVO* AND *HPLOW*) OR (*ENG2* AND *2.5L* OR (*ENG1* AND *EVO* AND *HPLOW*) OR (*L* AND (*ENG1* AND (NOT *EVO* OR (*ENG1* AND *EVO* AND *HPLOW*) OR (*ENG2* AND (*2.5L* OR *3.0L*))))) OR (*ENG3* AND (*1.6L* OR *2.0L*) AND *HPLOW*))))).

Items are implied (IF-THEN) with a feature variant combination as the IF condition, and the inclusion of the item as the THEN action. The group of items implied from a feature variant combination is commonly known as a Bill of Material (Bucki, 2015). The operational semantic of (IF-THEN) for production systems is known as *production rules* (Object Management Group, 2009). In this paper the implication of items is called *item usage rules*:

- IF (*GMAN* AND *ENG1*) THEN *ITM001*
- IF (*GMAN* AND (NOT (*ENG1*))) THEN *ITM002*

In this example, the vehicle configuration rules were presented as a list. The list is one common visualization method for configuration rules used during their inspection. The inspection of configuration rules is done during the development of the configuration rules. The development process for configuration rules will now be presented, in order to describe when and how inspection takes place.

1.1. Inspection of configuration rules

For mass customization, the development of configuration rules precedes the sales configuration, as shown in Figure 2. No product configurations can thereby be manufactured before the configuration rules set is complete, which generates

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

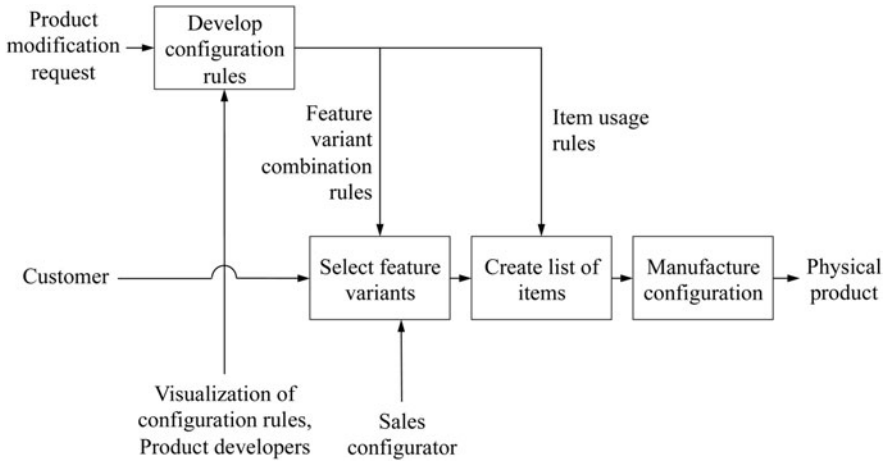

Fig. 2. Development of configuration rules precedes the arrival of customers and the manufacturing.

a high number of configuration rules for variant-rich products.

The process of developing configuration rules could be described with several activities, as shown in Figure 3. The authoring of new or modified configuration rules includes elicitation, interpretation, formalization, and implementation. The inspection, computation, and testing ensure that the configuration rules set is complete and correct. The release is when the configuration rules are frozen and made available to the sales and manufacturing departments. As can be seen in the figure, the inspection of configuration rules is a central activity, from which iterations can be both initiated and ended. The current aid during the inspection of configuration rules is the visualization tool of the product data management system. The visualization tool applies a certain visualization method for the configuration rules (e.g., lists, tables, matrices, graphs, trees etc.). This paper will use matrices as the visualization method in order to show the research results. The matrix-based visualization was first proposed as a concept by Bertin (1983).

The matrix-based visualization method is now explained by comparing it to a list-based visualization. A list is defined as several rows, which are not further divided into columns, as shown in Table 1.

The matrix does, however, have both rows and columns, and can thereby use symbols, (e.g., an *x*), for the relationships between rows and columns. The previously shown list-based item usage rules have in Table 2 been visualized with a matrix.

Inspection of rule-based representations is challenging for a product developer when the number of configuration rules is high, as well as when there are many feature variants in one single configuration rule. For instance, there are about 10^{21} possible vehicle configurations reported from a Renault product family (Astesana et al., 2010*b*), and 10^{103} vehicle

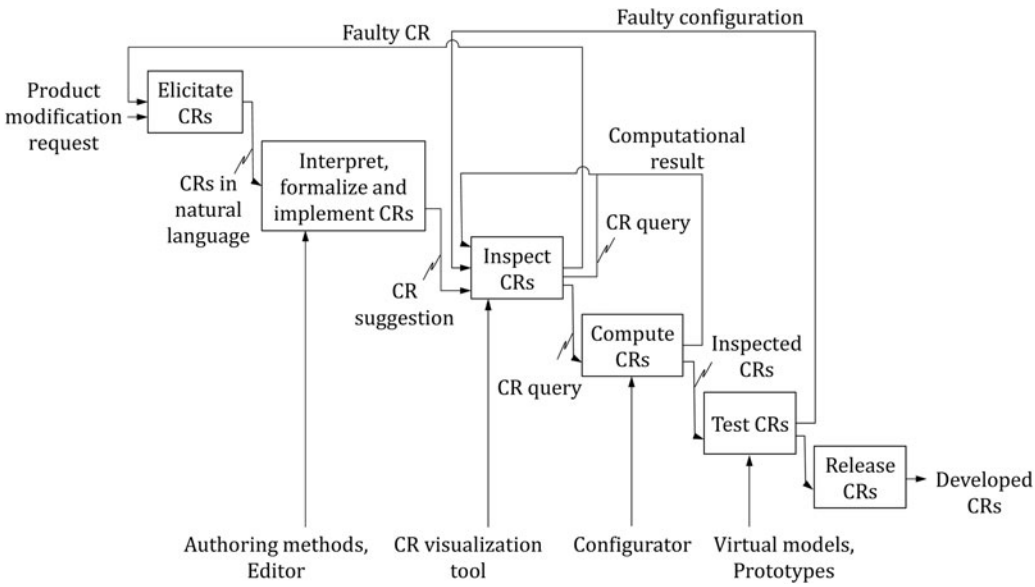

Fig. 3. Inspection of configuration rules is the activity where the majority of decisions about iterations is taking place.

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

Table 1. *Example of list-based visualization of item usage rules*

IF (*GMAN* AND *ENG1*) THEN *ITM001*
IF (*GMAN* AND (NOT (*ENG1*)) THEN *ITM002*

configurations from a Mercedes-Benz product family (Kübler et al., 2010). The vehicle configuration rules might forbid vehicle configurations that should be allowed, or they may allow vehicle configurations that cannot be assembled. Some of these problems can be discovered as late as on the manufacturing floor, which might result in costly manual adjustments or renegotiations of the order with the customer. To prevent a proliferation of errors from development to manufacturing, the configuration rules should be inspected before they are released to the sales and manufacturing departments. The identified research gap is the lack of examples for how the inspection of configuration rules can be supported by the solutions to various formulations of the CSP.

This paper proposes the modeling and solving of the CSP as an aid during the inspection of configuration rules. This is in line with researchers who emphasize the importance of visualizing configuration rules (e.g., Baumeister & Freiberg, 2010; Tidstam et al., 2012). The paper will show examples for how the product developers could take advantage of CSP variations during the inspection. The paper is therefore directed both to readers with a background in product development and to computer scientists interested in constraint satisfaction. There are three tasks where reasoning methods could aid the inspection of configuration rules, which gives the following research question:

> Which formulations of the CSP can make the inspection of configuration rules in propositional logic more efficient, when it comes to
>
> a. the reformulation of configuration rules,
> b. the testing of feature variant combinations, and
> c. the counting of item quantities from an item set?

1.2. Research approach

This research paper is one of several papers from a research project conducted in collaboration among three Swedish automotive companies. Numerous workshops identified the need for improved support for the visualization of configuration rules. The inspection activities that are addressed in this paper are identified by all three studied companies to have potential for improved computer support. A literature review was then conducted to compare the findings from the workshops with the already published literature on development of configuration rules. The literature review in Section 2 surveys the questions stated by product developers during the development of configuration rules. Then, described in Section 3, a use case scenario is described where the findings from the literature review are discussed in terms of the inspection of configuration rules. Section 4 describes a computational model for rule-based configuration: the computational model CSP. The computation model will be slightly modified for each addressed inspection activity, and those modifications are found in Section 5. The solution of the CSP will be presented together with a visualization of configuration rules in Section 6. The computational feasibility was evaluated and presented together with the case study. The conclusions are described in Section 7. The formulations of the CSP variations were developed by iteratively demonstrating their solutions in a matrix of configuration rules. This iteration was conducted internally among the coauthors, as well as externally with product developers at the case company. The evaluation of the research results has also been done with the usability tests of a visualization tool described in Tidstam et al. (2012). The usability tests focused on the visualization tool itself, and this paper is complementary describing the CSP variations that were used.

Table 2. *Example of matrix-based visualization of item usage rules*

Item ID	*GMAN*	*ENG1*	*ENG2*	*ENG3*
ITM001	X	X		
ITM002	X		X	
ITM002	X			X

2. LITERATURE REVIEW

The needs of product developers during inspection of configuration rules were first addressed in Sinz et al. (2003 by introducing formal methods. Questions from product developers during the inspection of configuration rules were listed and addressed. Similarly, Astesana et al. (2010*a*, 2010*b*) found questions from product developers when developing vehicle configuration rules. These questions have been categorized and summarized in Table 3. The categories used were *never allowed feature variant or item, testing feature variant combinations, reformulation of configuration rules*, and *counting quantities from item sets*. As can be seen in the table, there are multiple questions for each category. Two of the categories also refer to questions found at the two reviewed studies, which points at a coherent situation at the two automotive manufacturing companies Renault and Daimler. The use case scenario in the following section will discuss how these development questions are managed during the inspection of configuration rules.

Configuration problems can be formulated as *constraint problems* (Tsang, 1993). Constraint problems are problems with three characteristic attributes: *variables*, their *domains*, and *constraints*. Variables are objects that can take on a variety of values. The set of possible values for a given variable is called its domain. Constraints impose limitations on the values that a variable, or a combination of variables, may be assigned.

A *solution* or a *valid assignment* is an assignment of a single value from its domain to each variable such that no constraint is

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

Table 3. *Comparison between questions during development of configuration rules in propositional logic*

Question	Never Allow. Feat. Var./Item	Test. Feat. Var, Combin.	Reform. of Config. Rules	Count. Quant. From Item Sets
Are there feature variants that are not allowed for any allowed configuration? (Sinz et al., 2003)	X			
Are there items that are not included in any allowed configuration? (Sinz et al., 2003)	X			
For any given item, is there at least one allowed configuration for it? (Astesana et al., 2010*a*, 2010*b*)	X			
Is there at least one allowed configuration for a partial configuration? (Astesana et al., 2010*a*, 2010*b*)		X		
Given a list of partial configurations, does it represent every allowed configuration? (Astesana et al., 2010*a*, 2010*b*)		X		
Is it possible to remove any feature variant from a configuration rule without modifying the set of allowed configurations? (Astesana et al., 2010*a*, 2010*b*)			X	
Given a set of configuration rules, which is the smallest set of feature variants needed to their authoring? (Astesana et al., 2010*a*, 2010*b*)			X	
Are there allowed configurations with more than one item from a set of mutually exclusive items? (Sinz et al., 2003)				X
Is there an allowed configuration that does not have any item from a set of items? (Astesana et al., 2010*a*, 2010*b*)				X
Is there an allowed configuration that uses two or more items from a set of items? (Astesana et al., 2010*a*, 2010*b*)				X

violated. A problem may have one, many, or no solutions. A problem that has one or more solutions is said to be *satisfiable*; otherwise, it is *unsatisfiable*. Typical analysis of constraint problems is to determine whether a solution exists, finding one or all solutions, finding whether a partial instantiation can be extended to a full solution, and finding an optimal solution relative to a given cost function. Such tasks are referred to as *CSPs*.

CSPs for finite-domain variables belong to the set of NP-complete problems (Cook, 1971), and to date there is no algorithm known that can solve an arbitrary problem instance with a time complexity that is better than exponential in the size of the input (Hertli et al., 2011). However, if a problem instance possesses special structure, then the instance is polynomial-time solvable (Aspvall & Plass, 1979; Dowling & Gallier, 1984; van Maaren, 2000).

Configuration problems do not belong to any of the known polynomial-time solvable classes. Still, many solvers are able to solve the configuration problems described in this paper within a reasonable time, much faster than the theoretically predicted worst-case running time. An explanation for this discrepancy is highly technical and outside the scope of this paper (for an explanation, see Voronov, 2013).

Solving CSPs is typically done as a combination of several search-based algorithms, and these algorithms exploit the factored representation of the problem; that is, the problem is composed of variables each of which has a value within the variable domain (Russell & Norvig, 2003). General purpose heuristics are used to cut away large portions of the search space by identifying variable and value combinations that violate the constraints. This cutting away of the search space is what allows solvers to efficiently solve complex problems.

3. USE CASE SCENARIO FOR INSPECTION OF RULE-BASED CONFIGURATION

The actor in the use scenario is the product developer responsible for the engines. The task is to inspect the configuration rules for the engine, because there have been some recent new engine developments and there may be some necessary development of the configuration rules set. New feature variants for the new engines have already been established early in the vehicle development project, so there is no risk that feature variants are missing. In addition, before the engines were developed, the new items were established. However, there is a risk that the new items do not have any or have faulty item usage rules, or that the allowed feature variant combination rules do not align with what should be offered to the customers.

The use case scenario will now discuss the questions that were found during the literature review and exemplified in Table 3:

- **Never allowed feature variant/item** is a verification that is not done through inspection of the configuration rules

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

by a dedicated product developer responsible for a certain set of items. This question is rather a verification that can be executed by anyone for the complete configuration rules set. The question is therefore not considered in the use case scenario when the configuration rules are inspected.

- **Testing feature variant combinations** is a method for avoiding inspecting the configuration rules. The feature variant combinations are calculated from the feature variant combination rules. The product developer for engines could use this method for discovering if there are some feature variant combinations that are allowed, but should be forbidden. The opposite could of course also be the case, that there are some feature variant combinations that are forbidden, but should be allowed. If either of these two cases occurs, there are faulty configuration rules that should be found and updated. It is the product developer for engines who is responsible for detecting the need of modifications in the configuration rules set.
- **Reformulation of configuration rules** is a task that is performed either by the product developer or by a supporting configuration rule specialist. The inspection of the item usage rules and feature variant combination rules is heavily dependent on the visualization of configuration rules. When a certain set of configuration rules are visualized together, it may be beneficial to reformulate the configuration rules in order to make the configuration set coherent or for any other wish the product developer will have on the visualization. There are several variations of the CSP developed in this paper for the reformulation of configuration rules.
- **Counting quantities from item sets** is an analysis of the item usage rules. The product developer has to make sure that each allowed configuration also has the correct engine items. This is done by studying the item usage rules, in order to detect if there are any gaps between the item usage rules and the feature variant combination rules. These gaps typically occur if there are new feature variants introduced to the feature families that are included in the item usage rules for engines.

The use case scenario is exemplified further in the next section, where the CSP variations developed in this paper will also be described. A computation model will be created, because it is the basis for the vehicle configuration. Each CSP variation will be described together with a subsection showing how its solution could be implemented in a visualization of vehicle configuration rules. The time efficiency of those solutions to the CSP variations executed on vehicle configuration rules will be measured in the last section.

4. COMPUTATION MODEL FOR RULE-BASED CONFIGURATION WITH FEATURES AND ITEMS

A rule-based configuration with features and items can be transformed into a CSP (Tsang, 1993). A CSP is a triple of variables X, domains D, and constraints C. More formally, a CSP is a triple

$$\mathcal{P} = \langle X, D, C \rangle, \tag{1}$$

where $X = \langle x_1, x_2, \ldots, x_u \rangle$ is a u-tuple of variables, $D = \langle D_1, D_2, \ldots, D_u \rangle$ is a u-tuple of corresponding finite domains, and $C = \{C_1, C_2, \ldots, C_v\}$ is a set of constraints.

A constraint C_j is a pair $\langle R_{S_j}, S_j \rangle$, where R_{S_j} is a relation on the variables in $S_j = scope(C_j)$ and $scope(C_j) \subseteq X$ is the set of variables over which C_j is defined. In this paper, the configuration rules are restricted to propositional formulas over atomic propositions $x_k = v$, where $v \in D_k$.

This paper uses some concepts that should be formally defined (Table 4). A *configuration* is a function $f : X \rightarrow D$ (the function f has elements from X as arguments and elements from D as function values), which is defined for all $x_k \in X$ (for all x_k that belong to X). A complete assignment f is

Table 4. *Formal definitions of mathematical concepts*

Symbol	Description	Example
$X_F = \langle x_{F,1}, x_{F,2}, \ldots, x_{F,m} \rangle$	m-tuple of feature families	(*engine type, engine options, . . . , steering*)
$X_I = \langle x_{I,1}, x_{I,2}, \ldots, x_{I,n} \rangle$	n-tuple of items	$\langle$*ITM*001, *ITM*002, . . . , *ITEM*015$\rangle$
$D_F = \langle D_{F,1}, D_{F,2}, \ldots, D_{F,m} \rangle$	m-tuple of corresponding sets of feature variants	$\langle${*ENG*1, *ENG*2, *ENG*3}, {*EVO, HPLOW, HPMED, HPHIG*} . . . , {*L, R*}$\rangle$
$D_I = \langle \{\text{true, false}\} \rangle^{\lvert X_1 \rvert}$	Boolean domains for the items	$\langle${*true, false*}, {*true, false*}, . . . , {*true, false*}$\rangle$
$C_F = \{C_{F,1}, C_{F,2}, \ldots, C_{F,j}\}$	Set of feature variant combination rules	{IF (*GMAN*) THEN ((*R* AND (*ENG1* AND (NOT *EVO*) OR (*ENG1* AND *EVO* AND *HPLOW*) OR (*ENG2* AND *2.5L* OR (*ENG1* AND *EVO* AND *HPLOW*) OR (*L* AND (*ENG1* AND (NOT *EVO* OR (*ENG1* AND *EVO* AND *HPLOW*) OR (*ENG2* AND (*2.5L* OR *3.0L*))))) OR (*ENG3* AND (*1.6L* OR *2.0L*) AND *HPLOW*)), . . . }
$C_I = \{C_{I,1}, C_{I,2}, \ldots, C_{I,K}\}$	Set of item usage rules	{IF (*GMAN* AND *ENG1*) THEN *ITM001*, IF (*GMAN* AND (NOT (*ENG1*)) THEN *ITM002*, . . . }

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

allowed when its domain values for each variable forms a solution to the CSP. In other words, *allowed* configurations are the ones that satisfy all configuration rules from C_F. A *partial* configuration is a partial function $g : X \rightarrow D$ defined for variables $x_k \in Y \subseteq X$. (means for all x_k that belong to the subset Y of X). We will call a partial assignment *allowed* if and only if it can be extended to an allowed complete configuration; that is, there exists a function h defined for $X \backslash Y$ (means all elements in X excluding the elements in Y), such that the function values of g and h together form a solution.

In a rule-based configuration with features and items, a configuration is an assignment of feature variants from which a set of items is derived. Each feature family must be assigned exactly one feature variant. Feature families X_F and items X_I become variables, sets of feature variants D_F and the true or false values of items D_I become domains, and feature variant combination rules C_F and item usage rules C_I become constraints. More formally, the rule-based configuration with features and items is according to Eq. (2) a triple

$$\mathcal{P} = \langle X_F \oplus X_I, D_F \oplus D_I, C_F \cup C_I \rangle, \tag{2}$$

where $X_F = \langle x_{F,1}, x_{F,2}, \ldots, x_{F,m} \rangle$ is an m-tuple of feature families, $X_I = \langle x_{I,1}, x_{I,2}, \ldots, x_{I,n} \rangle$ is a n-tuple of items, $D_F = \langle D_{F,1}, D_{F,2}, \ldots, D_{F,m} \rangle$ is an m-tuple of corresponding sets of feature variants, $D_I = \{true, false\}^{|X_I|}$ is the Boolean domains for the items, $C_F = \{C_{F,1}, C_{F,2}, \ldots, C_{F,j}\}$ is a set of feature variant combination rules, and $C_I = \{C_{I,1}, C_{I,2}, \ldots, C_{I,K}\}$ is a set of item usage rules.

Note that the $\oplus$ symbol used here is to show that X_F only has the domain values from D_F and not D_I, even though the two domains exist in the CSP. The same is true for X_I, which can only have domain values from D_I. The number of items in X_I is given from the vertical lines symbolizing the magnitude, $|X_I|$. The number of $\{true, false\}$ tuples should correspond to the number of items in X_I. Finally, the symbol for union $\cup$ simply creates a configuration rules set consisting of both C_F and C_I.

The next section will describe the CSP variations developed for automated reasoning during the inspection of configuration rules.

5. VARIATIONS OF CSP ADDRESSING INSPECTION ACTIVITIES

The inspection activities will now be addressed as variations of the CSP. The addressed inspection activities are *testing feature variant combinations, counting quantities from item sets*, and *reformulation of configuration rules*.

5.1. Testing feature variant combinations

The testing of feature variant combinations is the testing of *partial* configurations. A partial configuration has fewer feature variants than a complete configuration. A partial configuration is allowed if it can be extended to at least one allowed complete configuration. The number of allowed feature variant combinations is much lower than the number of allowed product configurations. The testing of feature variant combinations is therefore a time-efficient alternative to the testing of complete configurations.

The complexity and interplay between configuration rules make it difficult to establish whether a new configuration rule is *correct*. For example, adding a configuration rule that forbids configurations that should normally be allowed is clearly undesirable. Product developers normally have some partial configurations they check if the configuration rules set is giving the expected results. This check can be facilitated if these partial configurations are stored as *reference configurations*. These configurations must always be possible to build, and if any of them becomes forbidden due to some added or modified configuration rule, then more thorough analysis is required. The reference configurations, as well as their counterpart *forbidden reference configurations*, are illustrated in Figure 4. Reference configurations can also be used as positive and negative examples for model-based diagnosis (Felfernig et al., 2004).

Verification of reference configurations P_i can be done by treating each reference configuration as an extra feature variant combination rule, and then checking for allowed configurations. No items or item usage rules need to be considered for verifying the reference configurations. More formally, the verification of a reference configuration is according to Eq. (2) defined as follows:

$$\mathcal{P} = \langle X_F \oplus \varnothing, D_F \oplus \varnothing, C_F \cup \varnothing \cup P_i \rangle, \tag{3}$$

where $X_F = \langle x_{F,1}, x_{F,2}, \ldots, x_{F,m} \rangle$ is an m-tuple of feature famies, $D_F = \langle D_{F,1}, D_{F,2}, \ldots, D_{F,m} \rangle$ is an m-tuple of corresponding sets of feature variants, $C_F = \{C_{F,1}, C_{F,2}, \ldots, C_{F,j}\}$ is a set of feature variant combination rules, and P_i is the reference configuration with index i.

By checking if this CSP has a solution, it is verified that a complete reference configuration satisfies a complete configuration. This can be done in time proportional to the number of configuration rules. An alternative approach would be to

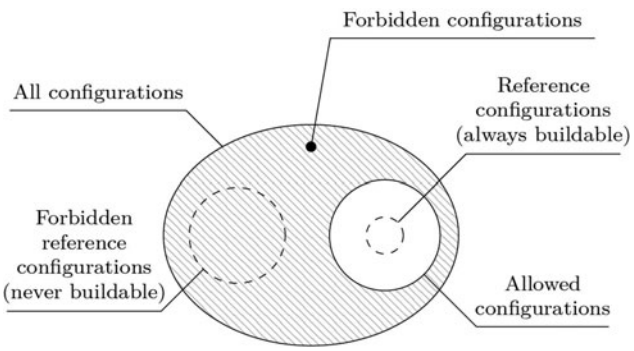

Fig. 4. Configuration space, with dashed mark-outs for configurations derived from reference configurations.

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

verify whether a partial reference configuration satisfies a new configuration rule. Verifying whether a partial configuration satisfies a set of configuration rules is an NP-complete problem, and to date the best known algorithms require an amount of time that, in the worst case, is exponential to the number of the variables that are not in the scope of the partial configuration.

5.2. Counting quantities from item sets

Item usage rules imply items from feature variants. Because there are no configuration rules between items, it is easy for item usage rules to be formulated in such a way that, for example, a configuration is missing some items. *At-least-one* condition must be satisfied for steering wheels, chassis, cabin, windscreen, and so on. Many of these examples also have a corresponding *at-most-one* condition; for example, a vehicle typically has only one steering wheel. Together, at-least-one and at-most-one conditions form *exactly-one* conditions. Exactly-one conditions can be illustrated as in Figure 5, which shows that every allowed configuration should have exactly one item. The issue is that there is no feedback from the product data management system about what kind of conditions the items meet.

How to avoid introducing problems when reformulating item usage rules, as well as how to discover opportunities to improve the structure of other configuration rules, is considered in the next subsection. To make sure that no two items from a set of items bound by an exactly-one condition ever appear in the same configuration, it is necessary to look at the item usage rules of these items. Exactly-one condition can be split into two conditions: at-most-one and at-least-one. We will look at these two conditions.

5.2.1. Verifying at-most-one condition

To find out whether a pair of items $u \in X_\mathrm{I}$ and $v \in X_\mathrm{I}$ can ever appear together in the same configuration, the following

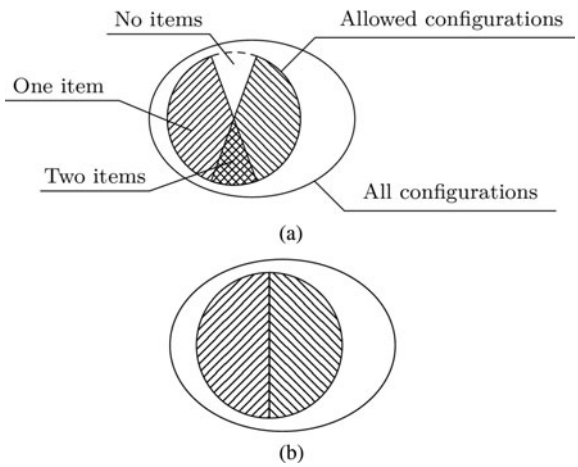

Fig. 5. Configurations for two items. (a) Without exactly one condition: some configurations with one item, some with two, and some with none. (b) With exactly one condition: all configurations have exactly one item, and there are no configurations with no item or several items.

configuration problem derived from Eq. (2) can be created that will have no allowed configurations if the items are mutually exclusive:

$$\mathcal{P} = \langle X_\mathrm{F} \oplus X_\mathrm{I}, D_\mathrm{F} \oplus D_\mathrm{I}, C_\mathrm{F} \cup \{C_\mathrm{I}^u, C_\mathrm{I}^v\}\rangle, \tag{4}$$

where $X_\mathrm{F} = \langle x_{\mathrm{F},1}, x_{\mathrm{F},2}, \ldots, x_{\mathrm{F},m}\rangle$ is an m-tuple of feature families, $X_\mathrm{I} = \langle u, v\rangle$ is items u and v, $D_\mathrm{F} = \langle D_{\mathrm{F},1}, D_{\mathrm{F},2}, \ldots, D_{\mathrm{F},m}\rangle$ is an m-tuple of corresponding sets of feature variants, $D_\mathrm{I} = \{true,false\}^2$ is the Boolean domain for the items u and v, $C_\mathrm{F} = \{C_{\mathrm{F},1}, C_{\mathrm{F},2}, \ldots, C_{\mathrm{F},j}\}$ is a set of feature variant combination rules, C_I^u is item usage rule for item u, and C_I^v is item usage rule for item v.

If there are allowed configurations in this new configuration problem, then items u and v can be included together for some allowed configuration of the original problem. If there are no allowed configurations in the new problem, then the items are mutually exclusive in the original problem.

For a mutually exclusive set with more than two items, a new configuration problem must be formulated for each pair of items in the set.

5.2.2. Verifying at-least-one condition

To make sure that at least one item from a set $S \subseteq X_\mathrm{I}$ is selected for every configuration, an additional configuration rule can be added that states that not all items from S should be false (not included for the configuration). Thus, if the corresponding CSP is satisfiable, there exists at least one configuration that includes none of the items from S. It is possible to formulate the following configuration problem derived from Eq. (2):

$$\mathcal{P} = \langle X_\mathrm{F} \oplus S, D_\mathrm{F} \oplus D_\mathrm{I}, C_\mathrm{F} \cup C_\mathrm{I} \cup \{\wedge_{s\in S}\neg s\}\rangle, \tag{5}$$

where $X_\mathrm{F} = \langle x_{\mathrm{F},1}, x_{\mathrm{F},2}, \ldots, x_{\mathrm{F},m}\rangle$ is an m-tuple of feature families, $X_\mathrm{I} = \langle x_{\mathrm{I},1}, x_{\mathrm{I},2}, \ldots, x_{\mathrm{I},n}\rangle$ is a n-tuple of items, S is a subset of X_I, $D_\mathrm{F} = \langle D_{\mathrm{F},1}, D_{\mathrm{F},2}, \ldots, D_{\mathrm{F},m}\rangle$ is an m-tuple of corresponding sets of feature variants, $D_\mathrm{I} = \{true,false\}^{|S|}$ is the Boolean domain for the items in S, $C_\mathrm{F} = \{C_{\mathrm{F},1}, C_{\mathrm{F},2}, \ldots, C_{\mathrm{F},j}\}$ is a set of feature variant combination rules, and C_I is the subset of item usage rules for the items in S.

5.3. Reformulation of configuration rules

Reformulation of configuration rules can greatly benefit from computation support. A product developer might want to visualize an item usage rule with a preferred set of feature variants (Tidstam et al., 2012). This cabe done in order to facilitate a better understanding of an item usage rule by showing it in combination with other feature variants. The reformulation only affects how the configuration rules are visualized to the product developer. Reformulating item usage rules does not change which items are applied for the allowed configurations.

The reformulation is similar to software code *refactoring*. Refactoring is a process of changing a software system in such a way that the external behavior of the code is not

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

altered, yet the internal structure of the code is improved (Fowler et al., 1999). It should be noted that refactoring was also introduced for knowledge bases (Baumeister et al., 2004), which are related to configuration rules. Reformulation was also proposed for feature models (Alves et al., 2006; Thüm et al., 2009).

5.3.1. *Reformulation of item usage rules*

There could be more than one item usage rule that could represent the same set of complete configurations for which an item is included. When that situation occurs, it is possible to reformulate the item usage rules for an item. Which item usage rule that was formulated during the development of configuration rules was hence nonambiguous. It is hence possible to use the item usage rules formulations interchangeably. We will with a CSP formulation determine if it is possible to add or remove feature variants from an item usage rule, while preserving whether an item is included or not included for each complete configuration. The formulation of the CSP will be described in text and then by an equation.

The first consideration for the CSP formulation is that it will be expressed for one single item. This has the following consequences:

- The variable X_{I} will only consist of one item, and not an n-tuple as for the previous CSP formulations. The CSP has then to be solved for each item individually, and the formulation will use the general item *i.*
- The domain for the item D_{I} is also only a single pair of true and false values, because only one item will be considered.
- The set of item usage rules C_{I} will only contain the item usage rule for item *i,* instead of the entire item usage rules set.

The next step is to create copies of the CSP containing only the single item *i.* This is necessary because the approach is to remain with the original formulation of the item usage rule but at the same time introduce the reformulated item usage rule. Copies are therefore created for all feature families, feature variants, and feature variant combination rules, as shown in Table 5. Shown in the table are also the copies of item *i*, its domain $\{true,false\}$, and its item usage rule C_{I}.

Another consideration for the formulation of CSP is the set of candidate feature families Y_{F}. The candidate set contains the feature families that are to be analyzed for possible reformulation of the item usage rule. For the set of candidate feature families, there is a need of synchronization between the original values and its copies. This synchronization is necessary, because we are in the final step going to compare item *i* with item. For example, when the candidate set of feature families Y_{F} contains the *engine type* with the value *ENG1* as feature variant, the original problem also has the value set to *ENG1*.

Finally, there is an additional constraint that the item *i* cannot be included in a configuration at the same time as item i'.

Table 5. *The extension of the original CSP formulation that also has copied values*

Description	Original	Copy
Feature families	X_{F}	X'_{F}
Items	$X_{\mathrm{I}} = \langle i \rangle$	$X'_{\mathrm{I}} = \langle i' \rangle$
Feature variants	D_{F}	D'_{F}
Boolean domains for items	$D_{\mathrm{I}} = \langle \{true,false\} \rangle$	$\mathrm{D}'_{\mathrm{I}} = \langle \{true,false\} \rangle$
Feature variant combination rules	C_{F}	C'_{F}
Item usage rules	C_{I} for item i	C'_{I} for item i'

If this configuration problem has allowed configurations, it indicates that the candidate set of feature families is not capable of uniquely determining the inclusion of the item, and thus cannot be used to reformulate the item usage rule of the item.

The formulation of the CSP will now be expressed more formally. Let X_{F} denote the tuple of variables that correspond to the original feature families. Let X'_{F} denote the tuple of variables that will correspond to the copies of the feature families. Let D_{F} and D'_{F} be the tuples of domains, for the original variables and the copies, respectively. Let C_{F} and C_{I} be the feature variant combination rules and item usage rules of the original feature families, and C'_{F} and C'_{I} be the copies. Let $Y_{\mathrm{F}} \subseteq X_{\mathrm{F}}$ be a subset of feature families. The following CSP will answer the question if Y_{F} contains feature families that could be reformulating the item usage rule C_{I} for item *i*:

$$\begin{aligned}\mathcal{P} = \langle & X_{\mathrm{F}} \oplus X_{\mathrm{I}} \oplus X'_{\mathrm{F}} \oplus X'_{\mathrm{I}}, D_{\mathrm{F}} \oplus D_{\mathrm{I}} \oplus D'_{\mathrm{F}} \oplus D'_{\mathrm{I}}, \\ & C_{\mathrm{F}} \cup C_{\mathrm{I}} \cup C'_{\mathrm{F}} \cup C'_{\mathrm{I}} \cup \{i \wedge \neg i'\} \\ & \cup \textstyle\bigwedge_{x_{\mathrm{F}} \in Y_{\mathrm{F}}} (x_{\mathrm{F}} = v) \leftrightarrow (x'_{\mathrm{F}} = v) \rangle, \end{aligned} \tag{6}$$

where $X_{\mathrm{F}} = \langle x_{\mathrm{F},1}, x_{\mathrm{F},2}, \ldots, x_{\mathrm{F},m} \rangle$ is an m-tuple of feature families, X'_{F} is a copy of X_{F}, $X_{\mathrm{I}} = \langle i \rangle$ is a 1-tuple of item *i*, X'_{I} is a copy of X_{I}, $D_{\mathrm{F}} = \langle D_{\mathrm{F},1}, D_{\mathrm{F},2}, \ldots, D_{\mathrm{F},m} \rangle$ is an m-tuple of corresponding sets of feature variants, D'_{F} is a copy of D_{F}, D_{I} is $\langle \{true,false\} \} \rangle$, D'_{I} is a copy of D_{I}, $C_{\mathrm{F}} = \{C_{\mathrm{F},1}, C_{\mathrm{F},2}, \ldots, C_{\mathrm{F},j}\}$ is a set of feature variant combination rules, C'_{F} is a copy of C_{F}, C_{I} is the item usage rule for item *i*, C'_{I} is a copy of C_{I}, Y_{F} is a subset of X_{F} and v is a value of x_{F}.

The next section will describe another type of analysis of item usage rules, which is not studying the formulation of single item usage rules, but a user-specified set of item usage rules.

5.3.2. *Are all configurations implying an item also implying another item?*

Let *u* and *v* be the two items constituting X_{I}. A product developer might be interested in the relationship between the items. For example, if *u* is selected, is it necessary to select *v*; that is, does *u* implying *v*? Does this implication hold

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

both ways? To check whether u implies v, it is possible to create a new configuration problem, which will have allowed configurations if and only if the implication is violated:

$$\mathcal{P} = \langle X_F \oplus X_I, D_F \oplus D_I, C_F \cup C_I^u \cup C_I^v \cup \{\neg(u \rightarrow v)\}\rangle, \quad (7)$$

where $X_F = \langle x_{F,1}, x_{F,2}, \ldots, x_{F,m}\rangle$ is an m-tuple of feature families, X_I is items $\langle u, v\rangle$, $D_F = \langle D_{F,1}, D_{F,2}, \ldots, D_{F,m}\rangle$ is an m-tuple of corresponding sets of feature variants, D_I is $\langle\{true,false\}\rangle^2$ are Boolean domains for item u and v, $C_F = \{C_{F,1}, C_{F,2}, \ldots, C_{F,j}\}$ is a set of feature variant combination rules, C_I^u is the item usage rules for item u, and C_I^v is the item usage rules for item v.

Note that $(\neg(u \rightarrow v)) = (u \wedge \neg v)$; that is, the problem will have allowed configurations if it is possible to select item u without selecting item v. To check whether the items imply each other, such new configuration problem can be constructed twice.

5.3.3. Can two items ever be selected together?

It might be useful to discover that for no allowed configuration two given items can be included together. Such knowledge might be useful, for example, when reformulating multiple independent items into a set of items bound by an exactly-one constraint. It is possible to create a new configuration problem, which will have allowed configurations if and only if the items can be included together for some allowed configuration of the original problem:

$$\mathcal{P} = \langle X_F \oplus X_I, D_F \oplus D_I, C_F \cup C_I^u \cup C_I^v \cup \{u \wedge v\}\rangle, \quad (8)$$

where $X_F = \langle x_{F,1}, x_{F,2}, \ldots, x_{F,m}\rangle$ is an m-tuple of feature families, X_I is items $\langle u, v\rangle$ $D_F = \langle D_{F,1}, D_{F,2}, \ldots, D_{F,m}\rangle$ is an m-tuple of corresponding sets of feature variants, $D_I = \{true, false\}^2$ are Boolean domains for item u and v, $C_F = \{C_{F,1}, C_{F,2}, \ldots, C_{F,j}\}$ is a set of feature variant combination rules, C_I^u is the item usage rules for item u, and C_I^v is the item usage rules for item v.

5.3.4. Are two feature variants equivalent?

Two feature variants can be considered equivalent if one feature variant implies the other and vice versa. This check can be done the same way as for items.

5.3.5. Are two feature families equivalent?

Two feature families x_F^i and x_F^j are equivalent if for each feature variant $v_k \in D_F^l$ of feature family x_F^i there is an equivalent feature variant $v_m \in D_F^j$, and the domain sizes are equal.

5.3.6. Is a feature variant redundant?

In the same way as for an item, it is possible to check whether a feature variant can ever be selected. If the feature variant is never possible to be selected, it is also redundant, and can be deleted without any substantial consequences. The CSP formulation analyzing the redundancy of a feature variant does not need to consider any items of item usage rules. Furthermore, the only addition to the CSP formulation is the forced selection of the possibly redundant feature variant. If the CSP is then evaluated as satisfiable, it would mean that the feature variant can be selected and is thereby not redundant. The for this research paper formulated CSP will only consider one feature variant at a time, and hence, the additional constraint is modified for each feature variant to be analyzed. The formulation of the configuration problem analyzing the feature variant v from feature family $x_{F,j}$ for redundancy is

$$\mathcal{P} = \langle X_F, D_F, C_F \cup \{x_{F,j} = v\}\rangle, \quad (9)$$

where $X_F = \langle x_{F,1}, x_{F,2}, \ldots, x_{F,m}\rangle$ is an m-tuple of feature families, $D_F = \langle D_{F,1}, D_{F,2}, \ldots, D_{F,m}\rangle$ is an m-tuple of corresponding sets of feature variants, $C_F = \{C_{F,1}, C_{F,2}, \ldots, C_{F,j}\}$ is a set of feature variant combination rules, and v is the possibly redundant feature variant from feature family $x_{F,j}$.

5.3.7. Is a feature family redundant?

If all but one feature variant can never be selected for some feature family, then the feature family can be seen as a constant, and can be removed, with the configuration rules simplified accordingly.

5.3.8. Is a feature variant combination rule redundant?

A configuration rule is redundant if the rest of the configuration rules imply it. Deleting a redundant feature variant combination rule will not have any substantial consequences, because there are other configuration rules that contain the same logic information. Expressing the redundancy of a configuration rule differently is to state that negating a redundant configuration rule would give a contradiction in the configuration rules set. This is also realized from logic discussion. A redundant configuration rule $C_{F,i}$ is implied by the rest of the configuration rules, which is stated with $(C_F \backslash C_{F,i}) \Rightarrow C_{F,i}$. This expression should always be true, and its negation should never be true. The negation is $\neg(C_F \backslash C_{F,i}) \Rightarrow C_{F,i}$, which simplifies to $(C_F \backslash C_{F,i}) \wedge \neg C_{F,i}$. The last expression is a configuration rules set that has been modified only by the negation for the redundant configuration rule. The formulation of a CSP for analyzing the feature variant combination rules will therefore not be satisfiable if a redundant configuration rule would be negated. This is the only modification to the original CSP, with the additional comment that no items or item usage rules need to be considered. The property can be checked as the absence of allowed configurations in the following CSP:

$$\mathcal{P} = \langle X_F, D_F, (C_F \backslash C_{F,i}) \cup \{\neg C_{F,i}\}\rangle, \quad (10)$$

where $X_F = \langle x_{F,1}, x_{F,2}, \ldots, x_{F,m}\rangle$ is an m-tuple of feature families, $D_F = \langle D_{F,1}, D_{F,2}, \ldots, D_{F,m}\rangle$ is an m-tuple of corresponding sets of feature variants, and $C_F = \{C_{F,1}, C_{F,2}, \ldots, C_{F,j}\}$ is a set of feature variant combination rules.

That the configuration rule is implied by the rest of the configuration rules is not always easy to use; there could be thou-

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

sands of configuration rules, while the redundant configuration rule can be implied by just a small subset of them. Together with the redundant configuration rule, that is why feedback for product developers should also contain an explanation of the redundancy, for example, as a (minimal) subset of configuration rules that are the reason for redundancy, and/or as a plain-text explanation. The minimal unsatisfiable subformula of the corresponding CSP can be used to extract the set of configuration rules that made the configuration rule in question redundant; see, for example, Büning and Kullmann (2009) and Liffiton (2009) for an introduction to minimal unsatisfiable subformula.

6. CASE STUDY FOR VISUALIZATION OF CSP SOLUTIONS

The solutions to the CSP variations will now be applied during the development of configuration rules. The three tasks that are addressed with CSP variations are *testing feature variant combinations, counting quantities from item s*ets, and *reformulation of configuration rules*. The last section will show a larger example, in order to test the scalability of the research results.

6.1. Testing feature variant combinations

The testing of feature variant combinations is the execution of the configuration rules to evaluate which feature variant combinations are allowed and which are forbidden. For products with a high number of feature variant combinations, the tests are conducted on partial configurations; that is, the test including number of feature variants is limited. This section will describe the use of partial reference configurations during the testing of feature variant combinations. Partial reference configurations are feature variant combinations that should be allowed or forbidden.

6.1.1. Before introducing CSP solutions

An experienced product developer was observed as he was performing a testing of feature variant combinations. The first step for the product developer was to execute the configuration rules in order to create a list of allowed partial configurations (see Table 6). The selection of feature variants was based upon the product developer's product configuration knowledge. The product developer then examined the list in order to find configurations that should be there or should not. In the table, the product developer was looking for the engine sizes available for engines without turbo. As shown in the table, there are two engine sizes (*1.2L* and *1.6L*), and they are both available for engines without turbo. The product developer knows which items have been developed, and this has to match which feature variant combinations that are allowed or forbidden. In this case example, the feature variant *without turbo* does not require any extra items and hence should be available for all offered engine size feature variants. The reference configurations could often be classified as facts, because they are repeated checks that should consistently hold for the product configurations. When the configuration rules are changed, the product developer repeats this testing of feature variant combinations in order to verify that the reference configurations still are allowed or forbidden according to his requirements.

Table 6. *List of allowed partial configurations*

Feature Family A	Feature Family B	Feature Family C
1.2L	Gasoline	Turbo
1.2L	Gasoline	Without turbo
1.6L	Gasoline	Turbo
1.6L	Gasoline	Without turbo
1.6L	Diesel	Turbo

6.1.2. After introducing CSP solutions

The testing of feature variant combinations are with the CSP solutions supported with an automated check of the partial reference configurations. The partial reference configurations that were verified by the product developer are stored as a list of test cases. The test cases could be introduced into the configuration rules list as additional configuration rules, as described in Eq. (3). An example of such list is presented in Table 7.

If this is done and there are no solutions to the CSP, there is at least one test case that is no longer allowed (P_1 or P_2). This means that there is no longer a visual inspection of the feature variant combinations necessary for these two partial reference configurations.

The discovery of the partial reference configurations have to, however, be captured during a manual inspection of the feature variant combinations, as was described in the Section 6.2.1. The capture of partial reference configurations is currently nonexistent at the automotive manufacturing company that was visited. The CSP solutions have thereby a potential for improving both time efficiency and quality of the configuration rules development process currently in use.

6.2. Counting quantities from item sets

The case that will be used to illustrate how items are counted from item sets is based on two items: *ITM001* and *ITM002*. This is a typical situation for the product developer, who has

Table 7. *List of partial reference configurations*

Index[a]	Partial Reference Configuration
P_1	*Without turbo* AND *1.2L*
P_2	*Without turbo* AND *1.6L*

[a]See Eq. (3).

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

to know if two items can be combined or not, which would be the prerequisite if an evaluation of clashes and functionality should take place. The possible combinations for items are alternative, could be used together, or should not be used at all for certain product configurations. This has previously been shown in Figure 5.

6.2.1. Before introducing CSP solutions

An example of how item usage rules could be visualized is shown in Table 8. Each row in the table contains one item usage rule. For example, the first row says that IF the customer ordered feature variant *1.6L*, *Turbo* and *Gasoline*, THEN the *1.6L* turbocharged engine item denoted *ITM001* should be included into the product assembly.

Before the introduction of CSP solutions, the product developers has to analyze both feature variant combination rules as well as the product model authorization rules in order to judge if there are any product configurations that does not have an item from *ITM001* or *ITM002*.

6.2.2. After introducing CSP solutions

The CSP variations suggested in this paper check whether there is any product configuration that has more than one item from an item set. This is achieved by introducing the item usage rules as feature variant combination rules, as described in Eq. (4). This means that for our case with *ITM001* and *ITM002*, the equation will have the additional feature variant combination rules:

$C_{\mathrm{I}}^{u} = (1.6L \text{ AND } Turbo \text{ AND } Gasoline)$, item usage rule for item *ITM001*, here indexed as u, and $C_{\mathrm{I}}^{v} = (1.6L \text{ AND } Diesel)$, item usage rule for item *ITM002*, here indexed v.

If there are allowed configurations in this new configuration problem, then *ITM001* and *ITM002* can be included together for some allowed configuration of the original problem. If there are no allowed configurations in the new problem, then the items are mutually exclusive in the original problem.

The result from the new configuration problem gave that there is a feature variant combination that is allowed, which then has neither *ITM001* nor *ITM002*. This was the feature variant combination shown in the last row in Table 9. It was found that the engines without turbo did not have an item for 1.6-L gasoline engines.

The item usage rule feedback should be provided to the product developers upon request; that is, it is a calculation to aid the product developers during the development of configuration rules. Not all item usage rules should have the *exactly-one* condition, which do not necessarily indicate an error in the configuration rules. The usefulness of the item usage rule feedback is however tremendous, because this evaluation is done manually today and it is a complex calculation that has to take all product configuration data for a product family into account.

Table 8. *Example of item usage rules*

Item ID	1.6L	Turbo	Gasoline	Diesel
ITM001	X	X	X	
ITM002	X			X

Table 9. *Example of how items can be counted from an item set*

Item ID	1.6L	Turbo	Without Turbo	Gasoline	Diesel
ITM001	X	X		X	
ITM002	X				X
No item?	?		?	?	

6.3. Reformulation of configuration rules

The reformulation of configuration rules can be requested from product developers, for example, when more information about the configuration rules is needed.

6.3.1. Before introducing CSP solutions

The reformulation of configuration rules was at the case company only taking place when the configuration rules were authored. The configuration rules were then not reformulated if they were not necessary because of modification in the configuration rules set. The reformulation of configuration rules is thereby a discussion between the product developers when the configuration rules are authored. The result from the discussion is an authoring of configuration rules, as in the example with item usage rules in Table 10. Often the item usage rules have been given an authoring after a negotiation of product developers' preferences.

6.3.2. After introducing CSP solutions

Some configuration rules might be implicit because of combined effects from several configuration rules. Some of these implicit configuration rules can be made explicit by reformulation. Equation (6) is used for finding feature families that can be introduced to the item usage rules during the reformulation. The results from the equation gave that there is a feature family with feature variant *City* that could be introduced, as shown in Table 11. The table shows a reformulation of the item usage rules for both *ITM001* and

Table 10. *Item usage rules formulated after a negotiation between the product developers*

Item ID	1.6L	Turbo	Gasoline	Diesel
ITM001	X	X	X	
ITM002	X			X

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

Table 11. *Reformulated item usage rules by introducing feature variant City*

Item ID	1.6L	Turbo	Gasoline	Diesel	City
ITM001	X	(X)	X		
ITM002	X			X	()

ITM002 from Table 10. The introduction of feature variant *City* is a reformulation of the item usage rules and consequently has introduced a pair of parenthesis in the matrix. This reformulation was possible because *City* is the only allowed feature variant from that family on all allowed configurations for *ITM002*. The introduction of *City* did not however result in any parenthesis, or reformulation, for *ITM001*, because this *City* may or may not be allowed in the allowed configurations with *ITM001*. The parenthesis for *Turbo* for *ITM001* was introduced because it is possible to take away the feature variant from the item usage rule without affecting for which configurations the item should be implied.

By using the CSP solutions, the reformulation of configuration rules can be shown upon request. It is then no longer required to actually reformulate the configuration rules, but possible reformulations can be shown in the visualization of configuration rules.

6.4. Larger industrial example

This section is applying the variations of CSP for reformulation of configuration rules as well as counting of items on a larger set of items. The case study showed examples with only two items, which is a common example when there are two alliterative components. There are however situations when there is a larger set of items that need to be analyzed. The example for larger set of items was selected based on a design engineer's, from the case company, request, and is shown in Table 12. The table provides 15 items, which are complemented with all the configurations that are allowed but does not imply any items from those. The potentially missing items are marked *No Item?* as well as question marks, as was previously shown in the case study. The potential reformulations of the configuration rules are marked with parentheses, as was also previously shown in the case study.

The feasibility of the example depends on its size limitations. The table for the 15 items has grown with additional six rows for configurations that are potentially missing items. This increase in number of rows depends on which items are selected to be visualized together, which then requires instructions for how to use the result from the CSP solutions. It is also shown in the table that the potential reformulation of configuration rules does not have any effect on the size of the table at all. This result is then promising for examples where the limited size of the table is very important.

6.5. Computational feasibility

This section will describe the time efficiency when the CSP variations were solved with the algorithm SAT4J. The inference engine used was thereby a SAT solver. The three configuration rules sets are from the three large automotive manufacturing companies. The data set from Renault Megane

Table 12. *Larger example for the visualization of constraint satisfaction problem solutions*

		A		B			C				D	
Item ID	Quantity	a1	a2	b1	b2	b3	c1	c2	c3	c4	d1	d2
No item?		?		?				?			?	
ITM001	2	X		X					X		()	
ITM002	2	X		X						X		
ITM003	2	X			X							
ITM004	2	X			X		X				()	
ITM005	2	X			X				X			
ITM006	2	X			X					X		
No item?		(?)				(?)	?					
No item?		?				?		?			?	
No item?		?				?		?				?
ITM007	2	X				X			X			
No item?		?				?				?	?	
No item?		?				?				?		?
ITM008	2		X	X					X		X	
ITM009	2		(X)	X					X			X
ITM010	2		X	X						X		
ITM011	2		X		X							
ITM012	2		X		X				X			
ITM013	2		X		X					X		
ITM014	2		X			X			X			
ITM015	2		X			X				X		

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

(Amilhastre et al., 2002) is available online (http://www.irit.fr/~Helene.Fargier/, but see the direct link at http://www.irit.fr/recherches/ADRIA/Documents/Fargier/Config/all.lp). The details of the data sets are presented in Table 13.

We benchmarked the time for verifying a partial configuration, the time to compute one allowed partial configuration when enumerating allowed partial configurations, the time to verify whether a given subset of feature families can be used to reformulate an item usage rule, and the time to analyze the effect of adding a configuration rule. Verifying allowed partial configurations and generating allowed partial configurations could be done incrementally (Een & Sörensson, 2004), reusing the solver instance; therefore, the timings are provided for incremental solving. Item usage rule verification requires creating a new problem, so the timing for item usage rule verification includes the time to generate the problem and to initialize the solver.

To benchmark the time to compute one allowed partial configuration when enumerating allowed partial configurations, we randomly generated subsets of feature families, containing from 3 to 30 feature families, and measured the time to generate 10 allowed partial configurations within each subset of feature families.

To benchmark the time for verifying a partial reference configuration, we randomly created 500 partial configurations containing from 3 to 30 feature families, and measured the time to verify whether a configuration is allowed or forbidden; Table 13 presents the average time for each data set.

Item usage rules were available only for product configuration data A, and there we measured the time to verify whether a given subset of feature families could be used to reformulate an item usage rule, by randomly selecting an item usage rule and generating random subsets of feature families, as well as by removing feature families from the original item usage rule. The measured times for reformulating the item usage rule include the time to generate the problem and initialize the solver.

Answering the questions presented in this paper took fractions of a second. The tests were performed on a desktop PC with 2-GHz processor and 8 GB of RAM. SAT4J (Le Berre & Parrain, 2010) was used as a CSP solver, with an extra layer of software for preprocessing the data, generating CSP instances, and interpreting solver answers. Generating partial configurations and enumerating configurations that become forbidden upon adding a configuration rule take even less time; one configuration can be generated in less than 0.01 s, using the incremental solving capabilities of SAT4J (Een & Sörensson, 2003, 2004).

These calculation times show that despite the problem formulations being NP-complete, practical times are short enough for interactive use by product developers.

7. CONCLUSIONS

The aim of this paper was to create CSP variations whose solutions automate previous manual tasks during the development of vehicle configuration rules. This paper addressed three development tasks taking place during the inspection of vehicle configuration rules: the reformulation of vehicle configuration rules, the testing of feature variant combinations, and the counting of item quantities from an item set. These three development tasks were addressed with CSP variations. The CSP variations were tested on vehicle configuration rules, and their feasible implementation in a visualization of vehicle configuration rules was proven. Thus, it was shown that the created CSP variations could actually be implemented into the development process of vehicle configuration rules.

The CSP variations proposed in this paper were shown to address similar challenges that were previously studied in Sinz et al. (2003) and Astesana et al. (2010*a*, 2010*b*). This paper should be seen as a continuation of these studies. It is a continuation because more variations of the CSP were created, and their implementation into the development process of vehicle configuration rules was described. The study of the development process is mentioned in the addressed research question. Which formulations of the CSP can make the inspection of configuration rules in propositional logic more efficient, when it

Table 13. *Feasibility study*

	Product Configuration Data		
	A	B	Renault Megane
Problem Properties			
Configuration rules	64161	9010	857
Feature families	511	217	102
All (allowed + forbidden) vehicle configurations	10^{150}	10^{85}	10^{50}
Whereof allowed vehicle configurations	10^{124}	10^{33}	10^{12}
Timing Results			
Time 1: partial configurations (s/verified configuration)	0.005	0.004	0.010
Time 2: partial reference configurations (s/verified configuration)	0.009	0.008	0.011
Time 3: item usage rule reformulations (s/verified reformulation)	0.75	NA	NA

https://doi.org/10.1017/S0890060415000487 Published online by Cambridge University Press

comes to the reformulation of configuration rules, the testing of feature variant combinations, and the counting of item quantities from an item set? The research question has three subquestions that were each addressed with development of CSP variations in this paper. The CSP variations developed are examples of how manual tasks can be automated. The time efficiency for inspection of vehicle configuration rules should therefore be improved, especially because we demonstrated that the automated computations take less than a second to complete.

This study has studied the inspection of vehicle configuration rules, which is one of several development process activities. The motivation for the chosen limitation to only include the inspection activity is that it takes place on existing vehicle configuration rules. Existing vehicle configuration rules enable testing of the CSP variations. Nevertheless, earlier process activities without any formalized vehicle configuration rules could also possibly benefit from the use of CSP solutions.

Our future work is to develop a demonstrator that allows product developers to take advantage of the CSP solutions as they are conducting their development activities. Thus far, examples of configuration rules have been supported with the CSP solutions and given to product developers at a large automotive manufacturing company. These examples have only been based on suggestions of suitable configuration rules from experienced product developers. What is then missed compared to the normal usage of a visualization tool is the importance of creating a selection of configuration rules that would be meaningful to be visualized. This selection of configuration rules may not necessarily be a trivial step, depending on how the companies group their configuration rules. Apart from that, the CSP variations were very well appreciated by the users, especially because the methods could simply be implemented as extra configuration rules, which made the result understandable and trustable.

ACKNOWLEDGMENTS

This work was carried out at the Wingquist Laboratory VINN Excellence Centre within the Area of Advance Production at Chalmers, supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA). The support is gratefully acknowledged.

REFERENCES

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., & Lucena, C. (2006). Refactoring product lines. *Proc. Int. Conf. Generative Programming and Component Engineering, GPCE '06*. New York: ACM Press.

Amilhastre, J., Fargier, H., & Marquis, P. (2002). Consistency restoration and explanations in dynamic CSPs—application to configuration. *Artificial Intelligence 135(1–2)*, 199–234.

Aspvall, B., & Plass, M. (1979). A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters 8(3)*, 121–123.

Astesana, J.-M., Bossu, Y., Cosserat, L., & Fargier, H. (2010*a*). Constraint-based modeling and exploitation of a vehicle range at Renault's: requirement analysis and complexity study. *Proc. Workshop on Configuration, ECAI 2010*. Amsterdam: IOS Press BV.

Astesana, J.-M., Cosserat, L., & Fargier, H. (2010*b*). Constraint-based vehicle configuration: a case Study. *Proc. Int. Conf. Tools With Artificial Intelligence, ICTAI 2010*. New York: IEEE.

Batory, D.S., Benavides, D., & Ruiz-Cortés, A. (2006). Automated analysis of feature models: challenges ahead. *Communications of the ACM 49(12)*, 45.

Baumeister, J., & Freiberg, M. (2010). Knowledge visualization for evaluation tasks. *Knowledge and Information Systems 29(2)*, 349–378.

Baumeister, J., Puppe, F., & Seipel, D. (2004). *Refactoring methods for knowledge bases. Engineering Knowledge in the Age of the Semantic Web*. Berlin: Springer–Verlag.

Bertin, J. (1983). *Semiology of Graphics: Diagrams, Networks, Maps*. Madison, WI: University of Wisconsin Press.

Bucki, J. (2015). *Bill of Materials*. Accessed at http://operationstech.about.com/od/glossary/g/BillMaterials.htm on May 1, 2015.

Büning, H.K., & Kullmann, O. (2009). Minimal unsatisfiability and autarkies. In *Handbook of Satisfiability* (Biere, A., Heule, M., van Maaren, H., & Walsh, T., Eds.), pp. 339–402. Amsterdam: IOS Press.

Chakraborty, R. (2010). *Knowledge Representations*. Accessed at http://www.myreaders.info/03-Knowledge_Representations.pdf on May 1, 2015.

Cook, S.A. (1971). The complexity of theorem-proving procedures. *Proc. ACM Symp*. New York: ACM Press.

Dowling, W.F., & Gallier, J.H. (1984). Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming 1(3)*, 267–284.

Een, N., & Sörensson, N. (2003). Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science 89(4)*, 543–560.

Een, N., & Sörensson, N. (2004). An extensible SAT-solver. *Theory and Applications of Satisfiability Testing 2919*, 502–518.

Felfernig, A., Friedrich, G., Jannach, D., & Stumptner, M. (2004). Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence 152(2)*, 213–234.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Boston: Addison–Wesley Professional.

Hertli, T., Moser, R.A., & Scheder, D. (2011). Improving ppsz for 3-sat using critical variables. *Proc. Int. Symp. Theoretical Aspects of Computer Science, STACS 2011*. Leibniz, Germany: Schloss Dagstuhl.

Junker, U. (2006). Configuration. In *Handbook of Constraint Programming* (Rossi, F., van Beek, P., & Walsh, T., Eds.), pp. 837–874. New York: Elsevier Science.

Krebs, T., Wolter, K., & Hotz, L. (2004). Mass customization for evolving product families. *Proc. Int. Conf. Economic, Technical and Organizational Aspects of Product Configuration Systems*, Copenhagen, June 28–29.

Kübler, A., Zengler, C., & Küchlin, W. (2010). Model counting in product configuration. *Proc. Workshop on Logics for Component Configuration, LoCoCo, 2010*. Sydney: EPTCS.

Le Berre, D., & Parrain, A. (2010). The Sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation 7*, 59–64.

Liffiton, M.H. (2009). *Analyzing infeasible constraint systems*. PhD Thesis. University of Michigan.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *National Conf. Artificial Intelligence, AAAI-90*. Boston: AAAI Press.

Object Management Group. (2009). *Production Rule Representation (PRR), International Standard (IEC) 61131-3*. Needham: Object Management Group.

Russell, S.J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Pearson Education.

Shortliffe, E. (1976). *Computer-Based Medical Consultations, MYCIN*. Amsterdam: Elsevier.

Sinz, C., Kaiser, A., & Küchlin, W. (2003). Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(1)*, 75–97.

Soininen, T., Tiihonen, J., Männistö, T., & Sulonen, R. (1998). Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12*, 357–372.

Thüm, T., Batory, D.S., & Kastner, C. (2009). Reasoning about edits to feature models. *Int. Conf. Software Engineering*, pp. 254–264. Los Alamitos, CA: IEEE.

Tidstam, A., Bligård, L.-O., Ekstedt, F., Voronov, A., Åkesson, K., & Malmqvist, J. (2012). Development of industrial visualization tools for validation of vehicle configuration rules. *Proc. Int. Symp. Tools and Methods of Competitive Engineering, TMCE'12*. Voorschoten: Emerald Eye.

Tsang, E.P. (1993). *Foundations of Constraint Satisfaction*. London: Academic Press.

van Maaren, H. (2000). A short note on some tractable cases of the satisfiability problem. *Information and Computation 158(2)*, 125–130.

Voronov, A. (2013). *On formal methods for large-scale product configuration*. PhD Thesis. Chalmers University of Technology.
Wehle, H.-D. (2011). *Cloud Billing Service.* Accessed at http://www.ibm.com/developerworks/cloud/library/cl-devcloudmodule/ on May 1, 2015.

Anna Tidstam is an IT Specialist in engineering systems at Thermo Fisher Scientific in München, Germany. She holds MS and PhD degrees from Chalmers University of Technology. Dr. Tidstam's PhD thesis focused on the development of vehicle configuration support tools and involved collaboration with Swedish, German, and French automotive companies.

Johan Malmqvist is a Chair Professor in product development at Chalmers University of Technology. His research addresses development methodologies and IT support for product development. Dr. Malmqvist's current research focuses on methods and tools for development of product–service systems, for product configuration, and for strategic development of methodologies and IT support for product development solutions. He has authored more than 100 publications in books, journal articles, and conference papers.

Alexey Voronov is a Senior Researcher at Viktoria Swedish ICT. He attained his PhD and MS degrees from Chalmers University of Technology. His PhD thesis was about formal methods for large-scale product configuration. Dr. Voronov's research interests include algorithms, formal methods, and complex systems.

Knut Åkesson is Associate Professor in the Department of Signals and Systems at Chalmers University of Technology. He holds a MS in computer science and technology from Lund Institute of Technology at the University of Lund, and a PhD in control engineering from Chalmers University of Technology. Dr. Åkesson's main research is in using formal methods on automated manufacturing systems. He has focused on using formal methods to solve manufacturing and product-related problems.

Martin Fabian is Professor of automation in the Department of Signals and Systems at Chalmers University of Technology. His research interests include formal methods for automation systems in a broad sense, merging the fields of control engineering, computer science, and production engineering. He has authored more than 100 publications and is codeveloper of the formal methods software tool Supremica.