Verification of Item Usage Rules in Product Configuration*

Alexey Voronov[1], Anna Tidstam[2], Knut Åkesson[1], and Johan Malmqvist[2]

[1] Dept. of Signals and Systems, Chalmers University of Technology
[2] Dept. of Product and Production Development, Chalmers University of Technology

Abstract. In the development of complex products product configuration systems are often used to support the development process. Item Usage Rules (IURs) are conditions for including specific *items* in products bills of materials based on a high-level product description. Large number of items and significant complexity of IURs make it difficult to maintain and analyze IURs manually. In this paper we present an automated approach for verifying IURs, which guarantees the presence of exactly one item from a predefined set in each product, as well as that an IUR can be reformulated without changing the set of products for which the item was included.

1 Introduction

Product configuration is widely used in automotive industry to better satisfy an individual customer needs while keeping the production costs down. Configuration is often split into two levels: high-level customer-oriented configuration and low-level manufacturing-oriented configuration [4]. Two levels not only abstract the customer from unnecessary details, they also facilitate the separation of relatively stable high-level features from rapidly-changing manufacturing *items*. The features are described by *families* that are instantiated in each product to a concrete *variant* from a predefined set. Since not all configurations result in buildable products, *variant rules* are used to define which configurations are valid and which are not. Due to rapidly changing nature of items, there are no rules between them to simplify the maintenance. There are, however, rules over variants and families, and there are *Item Usage Rules* (IURs) [13] that govern which items are selected with which variants.

The absence of item-to-item rules makes it possible to have inconsistencies in the configuration model hidden behind the complexity of variant rules and IURs. Consider a set of mutually-exclusive required items (SMI) where exactly one item from a SMI must be present in each product (SMIs are also called generic items [15]). An example could be that a car must have exactly one engine, and all

* 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.

L. Rivest, A. Bouras, and B. Louhichi (Eds.): PLM 2012, IFIP AICT 388, pp. 182–191, 2012.

items that represent different engines form a SMI. Since there are no item-to-item rules, exactly-one condition must be ensured through the combination of IURs and variant rules. It is thus easy to oversee a configuration that is valid according to the variant rules, but has no item from a SMI assigned to it. It is up to the engineers creating the rules to make sure that each product will have exactly one item from a SMI.

Configuration rules require maintenance over time: new items are introduced, old ones removed, and new families and variants are added. Sometimes it is necessary to reformulate an IUR in terms of a different set of families than the originally used. Due to the presence of variant rules, the same condition for item inclusion can be written differently, thus there could exist a number of equivalent IURs. For example, it could be possible to simplify an IUR without changing the configurations for which an item is included. Again, it is up to the engineer to ensure correctness of such IUR rewriting.

Even today verifying the correctness of configuration rules is to a great extent a manual process relying on the engineering experience and intuition. Historically exhaustive computer methods applied to verification were facing complexity problems, since configuration rules create constraints networks that are very complicated in a general case. Recent improvements in computer hardware and algorithms, especially in the areas of constraint satisfaction [10], made it possible to better assist design engineers in authoring, verification and analysis of the configuration constraints, e.g. [12,1,14]). A recent survey [3] lists different configuration areas that could benefit from algorithmic tools support. However, to the authors knowledge no algorithms for supporting the automated analysis of IURs have been presented.

The research questions addressed in this paper are the following:

- RQ1. How to help engineers in verifying IURs by ensuring that exactly one item from a SMI will be included in each valid configuration?
- RQ2. How to help engineers in authoring alternative IURs by verifying that an IUR can be rewritten in terms of a given subset of families without changing the valid configurations for which the item is included?

The contributions of this paper are a method to verify that exactly one item from a SMI is included in each valid product and a method to support engineers in authoring alternative IURs by verifying that alternative IUR describes the same set of valid configurations.

The paper is organized as follows. Section 2 presents a motivating example, Section 3 describes the verification and Section 4 concludes the paper.

2 Motivating Example

We will use a simplified car configuration example to illustrate the configuration and IURs and to show how different representations are possible. A formal treatment is given in Section 3.

Consider the families and variants shown in Table 1a, with variant rules shown in Table 1b.

Table 1. An example car configuration

(a) Families and variants

Family	Variants
Volume	1.6, 1.2
Turbo	yes, no
Sport	yes, no
City	yes, no
Fuel	gasoline, diesel

(b) Configuration rules for families

`if` Sport=yes `then` Volume=1.6
`if` Sport=yes `then`−and−only−`then` Turbo=yes
`if` Sport=yes `then` Fuel=gasoline
not (Sport=yes and City=yes)
not (Volume=1.6 and Fuel=gasoline and Sport=no)
`if` City=yes `then` Volume=1.2
`if` Fuel=Diesel `then` Volume=1.6

Table 2. IURs. In each row a condition for including an item is specified. For example, the first row specifies that if Volume=1.6 and Turbo=yes and Sport=yes and City=no and Fuel=gasoline then and only then item E16T should be included. For multi-row items (*E12* in this example), an item is included if and only if any of the rows is satisfied

Families and variants					
Volume	Turbo	Sport	City	Fuel	*Item(s)*
1.6	yes	yes	no	gasoline	*E16T*
1.6	no	no	no	diesel	*E16D*
1.2	no	no	yes	gasoline	*E12*
1.2	no	no	no	gasoline	*E12*

Table 2 shows a table representation of an example IURs for such car engine configuration, expressed over all of the families. In a real case with hundreds of families it is not feasible to express the IURs using all of the families, because the table will have to enumerate all valid configurations, and there could be 10^{20} and more valid configurations in industrial examples [2]. Luckily, many different configurations use the same item, and this introduces the first possibility to reduce the number of families used in IURs. Another possibility to remove families from the IURs is based on the fact that not all configurations are valid. Including non-valid configurations in the IUR does not change the configurations for which the item is used, but can reduce the number of families needed to express the IUR.

The IUR consists of two parts: the inclusion condition – a propositional formula over assignment of variants to the families, and the item itself.

In Table 2 each IUR have an item inclusion condition over families expressed in a Disjunctive Normal Form (DNF), that is, it is a disjunction (OR) of conjunctions (ANDs): it is a disjunction between the rows for the item, and a conjunction between the columns-families. There exist other ways to visually represent IURs, see e.g. [11] for an overview, but we use a simple table representation here.

The IUR for item named *E16T* (representing 1.6L turbocharged engine) can be written as logic expression as follows:

`if` Volume=1.6 and Turbo=yes and Sport=yes and City=no and Fuel=gasoline `then`−and−only−`then include` *E16T*

The rule for item *E12*, which is defined with two lines in the table, can be written as follows:

`if` (Volume=1.2 and Turbo=no and Sport=no and City=yes and Fuel=gasoline) OR (Volume=1.2 and Turbo=no and Sport=no and City=no and Fuel=gasoline) `then`−and−only−`then include` *E12*

2.1 Verifying Sets of Mutually-Exclusive Required Items (SMIs)

SMI defines a set exactly one item from which must be included in each product. Items *E16T*, *E16D* and *E12* form a SMI, thus each product and each valid configuration must have exactly one of these items. This property can be violated in two ways described below.

No Item Assigned. The example above satisfies this property, but it is easy to introduce a small change that will break it. Consider forgetting the last line in Table 2. Then there will be a car configuration with no engine.

More Than One Item Assigned. Representing IURs in DNF as in Table 2 has advantage of easy detection of the case when more than one item is being assigned to the same configuration: the items will simply end up on the same row in the table. However, removing the DNF requirement makes the table more compact, consider, for example, the IUR for item *E12*, which could be written as one line instead of two if we skip the DNF requirement and use a disjunction of variants in a cell, as in Table 3. In the row of the table, the cell for family City has two alternative values, which would correspond to the following expression:

`if` Volume=1.2 and Turbo=no and Sport=no and (City=yes OR City=no) and Fuel=gasoline `then`−and−only−`then include` *E12*

Table 3. IUR in non-DNF form (two variants for City)

Families and variants					
Volume	Turbo	Sport	City	Fuel	*Item(s)*
1.2	no	no	yes/no	gasoline	*E12*

With multiple variants per table cell it is not so easy to see whether multiple rows overlap in valid configurations or not (it is not a problem if they overlap

on non-valid configurations). An example of such overlap could be the two rows of Table 4, where the first row describes valid configurations of exactly one item, while the second row describes valid configurations of all three items. The first row of the table can be used as an IUR for *E12*, while the second row of the table cannot, since it includes valid configurations of all three items. Here, we rely on the correct IURs from Table 2 to highlight that the inclusion condition (written as a second row in the table) describes configurations of multiple items, and thus cannot be used in an IUR for an item in the SMI. However, when IURs are created, there is no such reference to rely on. It is desirable to be able to have an algorithm that could determine if the condition is fulfilled or not. Such algorithm is presented in Section 3.

Table 4. Overlapping IURs

Families and variants					
Volume	Turbo	Sport	City	Fuel	*Item(s)*
1.2/1.6	yes/no	yes/no	yes	gasoline/diesel	*E12*
1.2/1.6	yes/no	yes/no	no	gasoline/diesel	*E16T+E16D+E12*

2.2 Verifying Alternative IURs

Different logic expressions might describe the same set of valid configurations, thus an IUR can be written in terms of any of them. The difference between expressions might arise, for example, when the conditions describe the set in different ways. Conditions could also include different subsets of non-valid configurations without changing the set of valid configurations, this can lead to a simplification of the conditions. An example is the IUR for *E16T* and its four alternative formulations in Table 5, describing the same set of *valid* configurations.

However, not all subsets of families allow rewriting an IUR and keeping the same set of valid configurations. SMI can be used to illustrate this. Consider the families of Table 6. Item *E16T* is included for some valid configurations described by the first row, and for some configurations the items is not included. Thus, subset of families {Volume, City} is not enough to rewrite IUR for item *E16T*, but is enough, for example, to rewrite IUR for *E12*. We are interested in verifying that a candidate subset of families is enough to express some original IUR(s).

3 Automated Verification of Item Usage Rules

A complete valid configuration, or simply a valid configuration, is an assignment of variants to all families such that all configuration rules are satisfied. A *valid partial configuration* is an assignment of variants to a subset of families that can be extended to a complete valid configuration.

Table 5. IURs describing the same set of valid configurations but expressed differently

(a) Original IUR

Families and variants					
Volume	Turbo	Sport	City	Fuel	*Item(s)*
1.6	yes	yes	no	gasoline	*E16T*

(b) IUR with included non-valid configurations, keeping redundant columns that contain tautological disjunction of all variants

Families and variants					
Volume	Turbo	Sport	City	Fuel	*Item(s)*
1.6	yes/no	yes	yes/no	gasoline/diesel	*E16T*

(c) With included non-valid configurations, no redundant columns

Families and variants		
Volume	Sport	*Item(s)*
1.6	yes	*E16T*

(d) Alternative set of families

Families and variants		
Volume	Fuel	*Item(s)*
1.6	gasoline	*E16T*

Table 6. IURs based on families Volume and City. Note that configuration "Volume=1.6 and City=yes" is non-valid and thus excluded from the table.

Families and variants		
Volume	City	*Item(s)*
1.6	no	sometime *E16T*, sometime *E16D*
1.2	yes	*E12*
1.2	no	*E12*

An IUR condition covers a subset of the configurations. A condition can either cover all valid configurations, or cover none of them, or cover some of them. The condition can also cover non-valid configuration, that is those that can never appear in a valid product.

If there are two or more conditions that specify inclusion of the same item, they can be merged into one condition, so that each item will have exactly one, possibly complex, inclusion condition in the IUR:

`if` c_1 `then include` i_1
`if` c_2 `then include` i_1
...can be converted to...
`if` c_1 or c_2 `then include` i_1

To not create reasoning tools from scratch we will utilize well-established methods from constraint satisfaction to answer our verification questions.

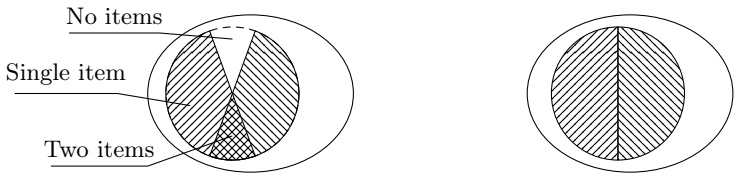
3.1 Constraint Satisfaction

Formally, a Constraint Satisfaction Problem (CSP) [9,8,10] is a triple $\langle X, D, C \rangle$, where X is a tuple of variables, D is a corresponding tuple of variable domains where $dom(x)$ denotes a domain for variable x, and C is a set of relations that determine which combinations of variables' values are valid. A solution to a CSP is an assignment of values for all variables such that all constraints are satisfied, or a certificate that such assignment is impossible. There exist a number of off-the-shelf CSP solvers including Gecode, Choco, JaCoP. CSP can often be converted into Boolean Satisfiability Problem (SAT), which is characterized by Boolean domains for all variables and constraints in Conjunctive Normal Form (conjunction of disjunctions). Both CSP [6] and SAT [12,5,16] have been shown applicable in configuration domain.

It is possible to encode product configuration as a CSP. Families become variables, variants become values, variant rules and IURs become constraints. Items become Boolean variables with values True and False, where True corresponds to the inclusion of the item into the product. In the rest of the section it will be shown how to encode verification of SMI properties and possibility to reformulate an IUR in terms of a given subset of families as CSP.

3.2 Verifying Sets of Mutually-Exclusive Required Items

If there are multiple items, then—depending on the inclusion conditions of each item—each valid configuration can have either: (i) no items assigned, (ii) exactly one item assigned, (iii) more than one item assigned. For the set of items that have the property that exactly one item from the set must be selected for each valid configuration, the first and the third cases are erroneous. The only correct case is the second one. This is illustrated in Figure 1.



(a) Incorrect: some configurations with one item, some with two, and some with none.

(b) Correct: disjoint and covers all configurations (ensures both *at most* and *at least* one item per configuration).

Fig. 1. Configurations for a SMI with two items

It is possible to automatically verify that a given set of items and their inclusion conditions indeed guarantees that each valid configuration will have exactly one item from the set.

Formulating as CSP. It is possible to create a CSP, solution to which will determine whether it is possible to have configurations with no items from a given set, as well more than one item from a given set.

Let $P = \langle X, D, C_0 \rangle$ be a CSP that describes valid configurations. Then to verify that items i_1 and i_2 never can be selected together for the same configuration, a constraint can be added to C_0:

$$P_{two} = \langle X, D, C \cup \{i_1 \text{ and } i_2\} \rangle$$

If P_{two} is satisfiable, then there is a configuration for which both items can be selected simultaneously, thus they do not form a mutually exclusive set. For more than two items in the set, it is possible to formulate this CSP for each pair individually, or create a cardinality constraint, which will be satisfied if two or more items from the set are selected.

Similar CSP can be formulated to verify that there are configurations with no items from the set $I = \{i_1, \ldots i_n\}$.

$$P_{none} = \langle X, D, C \cup \{(\text{not } i_1) \text{ and } (\text{not } i_2) \text{ and } \ldots \text{ and } (\text{not } i_n)\} \rangle$$

If P_{none} is satisfiable, then there is a configuration with no items from the set.

3.3 Verifying Alternative Formulations of an IUR

We will consider a way to automatically check that alternative formulation of the IUR is equivalent to the original in the sense that both of them require inclusion of the item for the same sets of valid configurations.

Formulation as CSP. For a fixed set of families it is possible to formulate a CSP that will encode that partial configurations for the set of families uniquely determine the inclusion or non-inclusion of the item. We can formulate the CSP as follows. We introduce every variable and constraint twice, thus having a clone of the problem. Then for the specified set of families, we force variants for families in the set to be equal between the clones, while variants for families outside of the set can be assigned independently between the clones. If this CSP is satisfiable, then there is a *partial* configuration for which an item can be either included or not included.

We have to create clones of the variables to allow two different complete configurations: we know in advance that there is no *complete* configuration where an item could be both included and not included.

Let X denote the tuple of variables that correspond to the original families. Let X' denote the tuple of variables that will correspond to the clones of the families. Let D and D' be the tuples of domains, for original variables and the

clones. Let C be the constraints encoding valid configurations and IURs of the original families, and C' be constraints encoding valid configurations and IURs on the clones. Let the function $clone : X \to X'$ be the mapping from original variables to the clones. Let $Y \subseteq X$ be a set of families. We are interested to test whether item i can be included and excluded with the same partial configuration, that is, whether there exist two complete configurations that become identical when projected on Y. Thus the following CSP will answer the question:

$$P_1 = \langle X \cup X', D \cup D', \\ C \cup C' \cup \{i \text{ and not } i'\} \cup \bigwedge_{x \in Y \subseteq X,\, v \in dom(x)} (x = v) \leftrightarrow (clone(x) = v) \rangle$$

where $(a \leftrightarrow b) \equiv ((a \text{ and } b) \text{ or } (\text{not } a \text{ and not } b))$. Equality sign denotes a selection of the value on the right-hand side for the variable on the left-hand side, and $X \cup X'$ denotes a tuple consisting of variables of X and X'.

If this CSP is satisfiable, then there is a partial configuration for which item can be included and not included, depending on variants for families in $X \setminus Y$, thus showing that Y is not enough to reformulate the IUR for i.

3.4 Empirical Evaluation

A prototype tool has been implemented to test the methods. Sat4j [7] was used as a CSP solver. Preliminary computational evaluation showed that on industrial data with 492 families and 10000 variant rules the verification of the SMI condition that two given items are never present simultaneously in a product can be done on average in under a second. Verification that a subset of families is enough to reformulate an IUR can be performed in under two seconds.

Table 7. Industrial test case

	# families in IUR	
Item ID	Original	Min found
i_{705}	4	1
i_{488}	3	3
i_{958}	3	1
i_{489}	4	1
i_{226}	4	2
i_{057}	4	1

Table 7 presents an industrial test case where reformulation of six IURs was investigated. Originally, the IURs used three to four families. Using the proposed method, it was found that some of the IURs can be shortened to a single family.

4 Conclusions and Future Work

We presented a method to verify correctness of IURs for items forming a SMI, namely that each valid product will indeed have exactly one item from a SMI. We also presented a method to verify that an IUR can be rewritten in terms of a given subset of families. Both verifications were encoded as CSPs and a tool based on Sat4j was implemented to solve the CSPs. These methods can be utilized to help design engineers in authoring and maintaining IURs.

As a future work, rewriting could be further automated by suggesting a minimal-size subset of families by utilizing constraint optimization procedures, with the positive verification as a constraint.

References

1. Astesana, J.M., Bossu, Y., Cosserat, L., Fargier, H.: Constraint-based Modeling and Exploitation of a Vehicle Range at Renaults: Requirement analysis and complexity study. In: ECAI 2010 Workshop on Configuration, pp. 33–39 (2010)
2. Astesana, J.-M., Cosserat, L., Fargier, H.: Constraint-based Vehicle Configuration: A Case Study. In: 22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, pp. 68–75. IEEE (2010)
3. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6), 615–636 (2010)
4. Haag, A.: Sales configuration in business processes. IEEE Intelligent Systems 13(4), 78–85 (1998)
5. Janota, M.: SAT Solving in Interactive Configuration (PhD thesis). Ph.D. thesis, University College Dublin (2010)
6. Junker, U.: Configuration. In: Handbook of Constraint Programming, ch.24, pp. 837–874. Foundations of Artificial Intelligence. Elsevier Science Inc. (2006)
7. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2 system description. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
8. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 8, 99–118 (1977)
9. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences 7, 95–132 (1974)
10. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming: Foundations of Artificial Intelligence. Elsevier (2006)
11. Sinz, C.: Comparing different logic-based representations of automotive parts lists. In: ECAI 2006 Workshop on Configuration, pp. 41–43 (2006)
12. Sinz, C., Kaiser, A., Küchlin, W.: Formal methods for the validation of automotive product configuration data. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(01), 75–97 (2003)
13. Tidstam, A., Malmqvist, J.: Information Modelling for Automotive Configuration. In: Proceedings of NordDesign 2010, Göteborg, Sweden (2010)
14. Tidstam, A., Malmqvist, J.: Authoring and verifying vehicle configuration rules. In: Proc. of 8th Intl. PLM Conference, Eindhoven, The Netherlands (2011)
15. van Veen, E.A.: Modelling Product Structures by Generic Bills-of-Materials. Elsevier Science Inc., New York (1992)
16. Voronov, A., Åkesson, K., Ekstedt, F.: Enumeration of valid partial configurations. In: Configuration Workshop at IJCAI 2011, Barcelona, Spain (2011)