Comparing Different Logic-Based Representations of Automotive Parts Lists

Carsten Sinz [1]

Abstract. Parts lists in the automotive industry can be of considerable size. For the Mercedes cars of DaimlerChrysler, for example, they consist of more than 30.000 entries for the larger model lines. Selection of the right parts for a particular product instance is complicated, and typically done via a logical formalism relating order codes with parts. To simplify part assignment, formalisms which use compact and concise formulae are required. We present and compare five different formalisms for such compact logical representations.

1 INTRODUCTION

In the automotive industry there is a persistent trend towards individually configured cars [1, 9]. This results in an enormous product variety that has to be coped with in sales, engineering, production, and after sales. Typically, the configuration of an individual car is accomplished on the level of *order codes*, which represent equipment options that a customer can select [3]. Such options include, among others, different engine types, wheel designs, interior and exterior colors, as well as car electronics like audio and navigation systems, and accessories like bike or ski carriers. As different equipment options may be mutually exclusive or require additional options, formalisms are needed to describe valid combinations. Moreover, automatic checking algorithms are needed to verify whether a customer's order is valid. A common way to describe these constrains is via logical formulae or rules [4, 5, 6, 7, 8, 10].

For each valid order (which satisfies all configuration constraints), in a second step, the right parts have to be selected. Mathematically speaking, this requires a mapping $M : \mathbb{P}(\mathcal{C}) \to \mathbb{P}(\mathcal{P})$ from sets of order codes to sets of parts (we denote by $\mathcal{C}$ the set of order codes, by $\mathcal{P}$ the set of parts, and by $\mathbb{P}(X)$ the powerset of X). Typically, the mapping can be broken down into a sequence of smaller mappings $M_1, \ldots, M_k$, one for each assembly position in the car. For an order $S \subseteq \mathcal{C}$, the required parts list $M(S)$ then is the collection of the required parts for all assembly positions, i.e. $M(S) = M_1(S) \cup \cdots \cup M_k(S)$. Moreover, the mapping for a position is often functional (but not necessarily total), such that the definition of M_i can be changed to $M_i : \mathbb{P}(\mathcal{C}) \to P$, and $M(S)$ becomes $\{M_i(S) \mid 1 \leq i \leq k\}$. We will assume such functional mappings in the rest of this paper.

There are different ways to represent these mappings, and choosing a suitable one is a non-trivial task, as it has to be concise, intelligible, as well as easily maintainable. In what follows, we restrict our attention to parts mappings for individual assembly positions, i.e. we are only interested in constructing the smaller mappings M_i. This makes a difference only from a practical point of view (sizes of considered parts sets), and has no influence on the proposed mathematical formalisms. It should also be noted that the mappings M_i not only have a reduced range (of zero or one in the functional case), but also typically depend only on a few dozen of codes, and thus can also be considered to possess a reduced domain.

[1] Johannes Kepler University, Linz, Austria, email: carsten.sinz@jku.at

2 PARTS LIST REPRESENTATIONS

We now turn to the question, how such parts list mappings M_i can be represented. Throughout this section, we use the following small example to illustrate the proposed methods: Assume three different order codes A, B, and C that influence an assembly position, and four different parts P1, ..., P4 which may be selected depending on the combination of selected codes (in reality there are up to a few dozen of codes and comparably many parts that have to be considered for each position[2]). In each valid configuration we assume that at least one of the codes A, B, C has to be present, and if A and B are selected, then C must also be present in the order. We further assume a parts mapping according to the following variant table:

variant	A	B	C	part
1	X			P1
2		X		P2
3			X	–
4	X		X	P3
5		X	X	P2
6	X	X	X	P4

Note that direct use of such a table is not feasible in practice, as, e.g., for a position depending on 20 codes it would contain up to 2^{20} lines.

2.1 Direct Propositional Encoding

The direct propositional encoding of the parts map uses propositional logic formulae (*parts rules*) that are associated with each part of a position. The parts rule is built upon the order codes, which are used as atomic propositions. To determine the matching part for a position, all the position's rules are evaluated based on the assignment induced by the customer's order (its characteristic function), and those parts for which the rule evaluates to true are selected. In our example, we would therefore obtain the following rule table:

parts rule	part
$A \wedge \neg B \wedge \neg C$	P1
$\neg A \wedge B$	P2
$A \wedge \neg B \wedge C$	P3
$A \wedge B \wedge C$	P4

[2] The largest position for Mercedes' E-Class limousines depends on 135 order codes and contains 27 different parts.

For the order $\{A, C\}$, e.g., rule $A \wedge \neg B \wedge C$ evaluates to true, and thus part P3 is selected.

However, this representation suffers from the drawback that negated codes have to be mentioned, too, which can cause a blow-up of the rules and make them harder to construct and maintain. It thus would be preferable to have a formalism that allows for more compact rules.

2.2 Propositional Encoding with Implicit Negations

The propositional encoding with implicit negations (IN) avoids specification of negated codes in rules and thus delivers a more compact encoding. Rules are computed from the variant table by building a *term* (conjunction of literals) for each row, removing negated codes from each term, and disjunctively composing terms that correspond to the same part. The resulting table is as follows:

IN parts rule	part
A	P1
$B \vee (B \wedge C)$	P2
$A \wedge C$	P3
$A \wedge B \wedge C$	P4

Now when computing the parts assignment for a particular order, an additional decoding step is required before evaluating the rules with the ordinary propositional semantics. This decoding works in two steps, inverting the encoding process:

1. First, all rules are converted to disjunctive normal form (DNF), such that they become disjunctions of terms (conjunctions).
2. Then, for each term, missing codes are added negatedly. Missing codes are codes that occur in the position, but not in the term.

After decoding, rules are evaluated as usual and the suitable part is computed as with the direct propositional encoding.

For the rule of part P2 (which is already in DNF), e.g., decoding delivers $(B \wedge \neg A \wedge \neg C) \vee (B \wedge C \wedge \neg A)$, which is equivalent to $B \wedge \neg A$ (the same as in the direct encoding). Care has to be taken in formulating the shortened IN rules in this formalism, however, as the absorption rule of Boolean logic is not valid any more. Thus, the rule for part P2 must not be simplified to B. Shortened rules of this formalism cannot only be derived from the variant table, but are supposed to be set up straightaway by the parts list maintenance personnel. Note, however, that the IN formalism requires one term for each row of the variant table for which a part is selected, which puts a natural limit on the compression capabilities of this formalism.

2.3 Propositional Encoding with Implicit Exclusions

A slight variant of the propositional encoding with implicit negations is that with implicit exclusion (IE). Like the former, it adds negated subformulae to terms of shortened rules in DNF and requires a decoding step to interpret rules; but in contrast to the former, it does not add missing literals, but rules of other parts, so-called exclusion formulae. The idea of exclusion formulae is to disambiguate part selection for overlapping rules by not assigning any part to the overlap. In more detail, the decoding step works as follows:

1. Compute the DNF of all rules, resulting in a set of conjunctions (terms) for each rule.
2. For each term T of each rule, conjunctively add negations of all terms S from other rules that are not subsumed by T, i.e. for which $S \not\subseteq T$ holds (S and T are regarded as sets of literals here).

As an example, consider the following table with IE rules:

IE parts rule	part
A	P1
$B \vee (B \wedge C)$	P2
$A \wedge C$	P3
$A \wedge B \wedge C$	P4

To decode the first IE rule (for part P1) we have to add negatedly all non-subsumed terms from rules of other parts. These non-subsumed exclusion terms are B, $B \wedge C$, $A \wedge C$ and $A \wedge B \wedge C$, such that the decoded rule for part P1 becomes $A \wedge \neg B \wedge \neg(B \wedge C) \wedge \neg(A \wedge C) \wedge \neg(A \wedge B \wedge C)$, which is logically equivalent to $A \wedge \neg B \wedge \neg C$. As a result, all overlaps with other parts are removed.

2.4 Propositional Encoding with Rule Priority

Another way to achieve more compact rules is by assigning them an evaluation order. This can be done by adding priorities (RP). Rules are then evaluated in order of decreasing priority. As soon as a rule matches, the decoding process is aborted and the respective part is selected. Using priorities we obtain a table like this for our example:

RP parts rule	priority	part
$A \wedge B \wedge C$	3	P4
B	2	P2
$A \wedge C$	2	P3
A	1	P1

Now, for a customer's order $\{A, C\}$, the rules are checked one by one in order of decreasing priority, starting with the rule for part P4, which has highest priority. As this rule does not match, we proceed to any rule with next highest priority (2 in our case), from which the one for part P3 matches. So this part is selected, and the decoding process is finished. A general rule of thumb to assign priorities—to rules consisting of only one term, at least—is to use the number of literals in the term. The RP formalism is used, e.g., in SAP Automotive.

2.5 Cascaded Conditions Algorithm

If priorities assigned in the RP formalism are all distinct, the formalism can be re-written in a more programmatic way, as it then is equivalent to a cascade (CC) of *if-then-else* expressions (or, alternatively, a *case* statement). Modifying the priorities in turn to 4, 2, 3 and 1 for the rows of our exemplary RP table, we obtain this program:

if $A \wedge B \wedge C$ **then** select(P4)
else if $A \wedge C$ **then** select(P3)
else if B **then** select(P2)
else if A **then** select(P1)

One problem with the *if-then-else*-cascades is that they are hard to maintain, especially if the rules and number of cases grow larger. Imagine, e.g., what would happen if $\{A, B\}$ became a valid order that selects no part? Which rules have to be changed in which way then? Maintenance can thus become a non-trivial task.

2.6 Best-Fit Algorithm

At last, we want to present an algorithm that avoids ordering of parts rules for evaluation, but still keeps advantageous properties of the priority-based approach and combines them with ideas from the IE formalism. It works by computing the quality of how good a rule fits

to a customer's order, and selects the best fitting one (BF). Different fitness (or quality) measures are possible, but we only present one that maximizes the number of matching literals. It works as follows:

1. Compute DNFs for all rules, giving a set of terms for each rule.
2. For each term that matches the order (i.e. evaluates to true), compute the number of literals that coincide with the order (matching positive literals that occur in the order as well as negative literals not occuring in the order are counted); this is the fitness measure.
3. If there is exactly one matching term with highest fitness measure, the corresponding part is included into the parts list. Otherwise (i.e. if no or more than one term with highest fitness measure matches), an ambiguity exists, and no part is chosen.

Consider again our examplary table, now with BF parts rules:

BF parts rule	part
A	P1
$B \vee (B \wedge C)$	P2
$A \wedge C$	P3
$A \wedge B \wedge C$	P4

For the order $\{A, C\}$ there are two matching terms, namely A and $A \wedge C$. The latter's fitness measure is 2, whereas the former's is 1. Thus part P3, corresponding to parts rule $A \wedge C$, is chosen.

3 COMPARISON

We now want to compare the aforementioned formalisms, starting with the IN and IE encodings. They seem quite similar, but differences become discernible on even small examples. Consider two parts rules, A and $B \wedge C$:

IN / IE parts rule	IN decoding	IE decoding	part
A	$A \wedge \neg B \wedge \neg C$	$A \wedge (\neg B \vee \neg C)$	P1
$B \wedge C$	$\neg A \wedge B \wedge C$	$\neg A \wedge B \wedge C$	P2

If we visualize the IN / IE part rules in a Venn diagram (Fig. 1 left, red/darker for the first, green/lighter for the second parts rule), we see that there is an overlap between the rules for an order containing all three codes A, B and C. The IN encoding (Fig. 1 middle) selects part P1 only if none of the codes B and C is present, whereas the IE decoding (Fig. 1 right) selects a part for all but the overlap, which is perhaps the more natural interpretation.

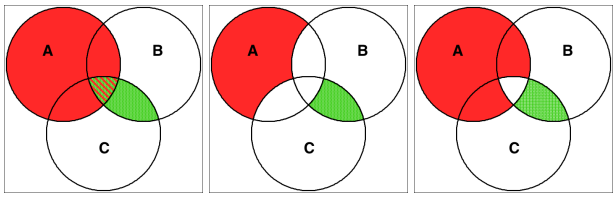

Figure 1. Different interpretations of IN and IE encoding. Left: IN / IE rule; middle: IN decoding; right: IE decoding.

Turning our attention now to all five logical parts list representations, we want to compare them regarding compactness of representation, intelligibility and ease of maintenance.

property	IN	IE	RP	CC	BF
compactness	–	+	+	+	+
intelligibility	0	0	0	+	0
ease of maintenance	–	–	0	–	0

In the encoding with implicit negation (IN) compactness suffers due to the fact that it requires one term for each entry of the variant table. This is not the case for all other encodings, which thus are more concise. Turning to intelligibility, all encodings should be comprehensible after some practice. However, the program-like encoding CC is perhaps the easiest to grasp. Maintaining a (large) logic-based parts list is not simple. This is especially the case for the IN encoding with its resulting bulky rules, but also for the IE encoding, where it might become hard for large rules to figure out the occurring overlaps. The same holds for the CC encoding, where many rules may have to be modified when inserting a new variant.

4 RELATED WORK

In knowledge representation similar problems like those of this paper arise. The most frequently proposed solution is that of assigning priorities (or weights) to rules [11]. Other related formalisms are negation-as-failure (cf. Prolog) or the stable model semantics [2].

5 CONCLUSION

We have presented five different ways to compactly represent logic-based parts lists. The relevance of these formalisms stems from the fact that they are already in practical use at different automotive companies. However, maintenance of logic-based parts lists is a complicated task that requires a thorough understanding of the basic logical formalism. Most of the methods presented in this paper become much more apprehensible when they are accompanied by tool support. In the IE or BF formalisms, e.g., a tool that shows rule overlaps would be very helpful.

In general, we take up the position that rule compilation and maintenance should be considered a programming task. As such, it could benefit from established software engineering methods like coding style-guides, testing or verification.

REFERENCES

[1] S. M. Davis, *Future Perfect*, Addison-Wesley, 1987.

[2] M. Gelfond and V. Lifschitz, 'The stable model semantics for logic programming', in *Proc. 5th Intl. Conf. on Logic Programming*, pp. 1070–1080. The MIT Press, (1988).

[3] A. Haag, 'Sales configuration in business processes', *IEEE Intelligent Systems*, **13**(4), 78–85, (July/August 1998).

[4] D. Mailharro, 'A classification and constraint-based framework for configuration', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, **12**(4), 383–397, (1998).

[5] D.L. McGuinness, 'Configuration', in *The Description Logic Handbook*, eds., F. Baader, D. McGuinness, P. Nardi, and P. Patel-Schneider, 397–414, Cambridge University Press, (2003).

[6] S. Mittal and F. Frayman, 'Towards a generic model of configuration tasks', in *Proc. of the 11th Intl. Joint Conf. on Artificial Intelligence*, pp. 1395–1401, Detroit, MI, (August 1989).

[7] D. Sabin and E.C. Freuder, 'Configuration as composite constraint satisfaction', in *Proc. Artificial Intelligence and Manufacturing Research Planning Workshop*, ed., G.F. Luger, pp. 153–161, Albuquerque, NM, (1996). AAAI Press.

[8] D. Sabin and R. Weigel, 'Product configuration frameworks – a survey', *IEEE Intelligent Systems*, **13**(4), 42–49, (July/August 1998).

[9] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin, 'Formal methods for the validation of automotive product configuration data', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, **17**(1), 75–97, (January 2003).

[10] M. Stumptner, 'An overview of knowledge-based configuration', *AI Communications*, **10**(2), 111–125, (1997).

[11] R.J. Waldinger and M.E. Stickel, 'Proving properties of rule based systems', *Intl. J. Software Engineering and Knowledge Engineering*, **2**(1), 121–144, (1992).