An Improved Constraint Ordering Heuristics for Compiling Configuration Problems

Benjamin Matthes and **Christoph Zengler** and **Wolfgang Küchlin**[1]

Abstract. This paper is a case study on generating BDDs (binary decision diagrams) for propositional encodings of industrial configuration problems. As a testbed we use product configuration formulas arising in the automotive industry. Our main contribution is the introduction of a new improved constraint ordering heuristics incorporating structure-specific knowledge of the problem at hand. With the help of this constraint ordering, we were able to compile all formulas of our testbed to BDDs which was not possible with an arbitrary constraint order.

1 INTRODUCTION

Since the early 80s, product configuration systems have been among the most prominent and successful applications of AI methods in practice [15]. As a result computer aided configuration systems have been used in managing complex software, hardware or network settings. Another application area of these configuration systems is the automotive industry. Here they helped to realize the transition from the mass production paradigm to present-day mass customization.

Besides CSP encodings [1] also propositional encodings [10] of configuration problems proved to be a viable alternative in the automotive industry. Specific queries to the configuration base can then be answered by a decision procedure for propositional logic, e.g. in many cases SAT solvers. Although modern SAT solvers prove to be very efficient in answering such queries, there are two major drawbacks: (1) Since decidability of a propositional formula is NP-hard, SAT solvers cannot guarantee certain runtime requirements required in online configuration applications; (2) there are some types of queries that cannot be handled by a SAT solver efficiently, e.g. restriction, model enumeration, or model counting. One approach to circumvent these limitations is the use of knowledge compilation.

The basic idea of knowledge compilation is to distinguish two phases: (1) an offline phase in which a given formula is compiled into the respective compilation format and (2) an online phase in which we query the compilation. Usually the offline phase is still NP-hard, but once compiled, there are a number of interesting polynomial time operations on the compilation. Well-known knowledge compilation formats for propositional logic are e.g. BDDs [3] or DNNFs [5].

For this paper we chose BDD as compilation format. Its use in configuration problems is well-studied [7, 11]. Hadzic et al. [7] focus on minimizing the final BDD for shorter response times; Narodytska et al. [11] try to establish a good static variable ordering for BDD compilation of configuration problems. They also present a constraint ordering based on the constraint graph. In contrast, in this paper we present an ordering of the constraints based on some structure knowledge of the problem which is not always deducible from the constraint graph. In most cases our test instances could only be compiled into BDDs with this new constraint ordering. With an arbitrary ordering we exceeded space or time limits.

In section 2 we will introduce propositional configuration problems and present the reader an overview of binary decision diagrams and some important properties. Section 3 shortly describes our test instances from the automotive industry. Our main contribution lies in section 4. We present an ordering of the constraints of the configuration problem with the help of which we could compile all formulas to BDDs.

[1] Symbolic Computation Group, WSI Informatics, Universität Tübingen, Germany, email: [matthesb, zengler, kuechlin]@informatik.uni-tuebingen.de

2 PRELIMINARIES

2.1 Configuration problems

2.1.1 *Propositional configuration problems*

We use the definition of a configuration problem as given in [7, Definition 1]: a configuration problem is a triple $(\mathcal{V}, D, \Psi)$ where $\mathcal{V}$ is a set of variables $x_1, x_2, \ldots, x_n$, D is a set of their finite domains $D_1, D_2, \ldots, D_n$ and $\Psi = \{\psi_1, \psi_2, \ldots, \psi_m\}$ is a set of propositional formulas (constraints) over atomic propositions $x_i = v$ where $v \in D_i$, specifying conditions that the variable assignments have to satisfy. A *valid configuration* is an assignment α with $\mathrm{dom}(\alpha) = \mathcal{V}$ such that $\alpha \models \bigwedge_{\psi \in \Psi} \psi$, i.e. all constraints hold.

In this paper we consider the special case where we have only propositional variables in $\mathcal{V}$ and hence $D_i = \{1, 0\}$ for all $1 \leq i \leq n$. The set $\mathcal{O}$ is the finite set of all configuration options for a product. Each variable $x_o \in \mathcal{V}$ represents a configuration option $o \in \mathcal{O}$. The variable x_o is assigned to 1 if the option o is chosen, otherwise it is assigned to 0. Following this course, the resulting formulas $\psi \in \Psi$ are propositional formulas and hence $\varphi = \bigwedge_{\psi \in \Psi} \psi$ is a propositional formula describing all valid configurations. We will also refer to φ as *product overview formula (POF)* [10].

Remark. The restriction of the variables $x \in \mathcal{V}$ to propositional variables does not limit the expressiveness of our problem description. Since the domains D_i are finite and we only allow atomic propositions of the form $x = v$, we can use a reduction [4] from equality logic to propositional logic.

2.1.2 *Structure of configuration problems*

In many application domains (including the automative product configuration), we can divide the set of constraints Ψ in three parts:

Unit Constraints Ψ_U constraints concerning only a single variable. These constraints enforce or forbid the selection of a single option.

Cardinality Constraints Ψ_{CC} Constraints enforcing the selection of a certain number of options. In most cases the selection of exactly one option or at most one option is enforced.

Dependencies Ψ_D Constraints describing the dependencies between two or more options. These constraints are used to describe complex domain specific configuration knowledge.

Example. In the automotive industry we have the following examples for the aforementioned constraint sets:

- Ψ_U: Necessary or forbidden options in a production series of cars. E.g. 'EPS must be chosen in this series' or 'automatic transmission is not available for this series'.
- Ψ_{CC}: Enforcement that only one option from a certain option-family can be chosen at the same time. E.g. 'only one steering wheel in a car', or 'at most one navigation system'.
- Ψ_D: Description of complex dependencies in a car. E.g. 'navigation system enforces also board computer and forbids radio'.

2.2 Ordered binary decision diagrams

A binary decision diagram [3] is a directed acyclic graph which represents a propositional formula. Each inner node is labeled with a propositional variable and has two outgoing edges for negative and positive assignment of the respective variable. The leaves are labeled with 1 and 0 representing *true* and *false*. An assignment is represented by a path from the root node to a leaf and its evaluation is the respective value of the leaf. Therefore all paths to a 1-leaf are valid models for the formula. Ordered reduced BDDs (ROBDDs) are a subset with additional restrictions for the BDDs. Ordering guarantees the same variable ordering on all paths through the BDD; Reduction guarantees that equivalent subtrees of the BDD are compactified and redundant nodes are deleted. A ROBDD is a canonical representation of a propositional formula wrt. to a variable ordering, meaning the ROBDD of a formula is unique. From now on we will refer to ROBDDs simply as BDDs. Figure 1 presents the BDD for the formula $(x_1 \leftrightarrow x_2) \vee x_3$ with the variable ordering $x_1 < x_2 < x_3$. Solid edges represent the positive assignment, dashed edges the negative assignment.

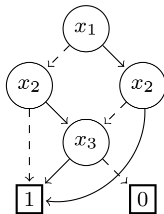

Figure 1. BDD for $(x_1 \leftrightarrow x_2) \vee x_3$ with ordering $x_1 < x_2 < x_3$

Once compiled, BDDs allow a large number of polynomial time operations on the represented formula. Among them are: satisfiability, general entailment, restriction or equivalence. Since satisfiability is a polynomial time operation on BDDs, it is obvious, that it is NP-hard to transform a given Boolean formula into a BDD. The size (number of nodes) of a BDD is strongly dependent on the variable ordering. There are many examples where bad orderings produce exponential size BDDs, whereas a good ordering produces a linear size BDD. So finding a good variable ordering is a crucial task in the compilation phase. Finding an optimal variable ordering is an NP-complete problem [2]. Different reordering heuristics for BDDs will be reviewed in section 2.2.1.

Since our input formulas are in CNF, the usual procedure of compiling the BDD is to generate BDDs for each clause and conjoin them. Here, the order in which the clauses are conjoined plays an important role. We will discuss the impact of this clause/constraint ordering in section 2.2.2.

2.2.1 Reordering heuristics

As already mentioned, finding the optimal variable order for a BDD is NP-complete. Modern BDD compilers use different heuristics to find a good variable ordering while compiling. We will present some of these heuristics which proved to be of interest for our real world applications.

The *sifting algorithm* by Rudell [14] is the foundation of various reordering heuristics. It is based on finding an optimum for each variable assuming all other variables remain fixed. Each variable is considered in sequence, beginning with the variable with most occurrences. The currently selected variable is sifted (moved) sequentially to both ends of the variable ordering and is finally fixed to the optimum position wrt. the size of the BDD. All variable movement can be done by a series of adjacent variable swaps. Swapping a variable with its direct predecessor or successor does *not* affect levels other than those of these two variables and therefore depends only proportionally on the size of the respective levels. This sifting process is repeated for each variable, in order of their occurrences. It is notable that the BDD size can increase heavily during sifting.

The sifting algorithm can be extended to a *symmetric sifting* [13], where symmetric variables (variables, that can be interchanged without changing the Boolean function) are kept close together. Symmetric sifting again can be generalized to *group sifting* [12]. Here, symmetry situations that go beyond the symmetry of two variables can be treated specially.

A different approach was suggested by Fujita et al. [6] and Ishiura et al. [8]. Instead of searching the optimal position of a variable in the whole variable ordering, the search space is restricted to a small *window*. Each variable is considered in sequence and permuted inside a window of size k. If x_i is considered and window size is 3, x_i, x_{i+1} and x_{i+2} have to be permuted. All $k!$ possibilities of arranging variables are exhaustively searched. After testing all permutations, the best one wrt. BDD size is used. The process is repeated for each variable. Due to the rapid growth of the faculty function, this approach is only practical for window sizes up to 5. Generally it performs better than sifting, but may not be able to overcome local minima.

For further comparison two random based algorithms have been used. The *random* variant randomly selects pairs of variables and transposes them with adjacent swaps. The best position wrt. BDD size is used. This step is repeated n times for n variables. The *random pivot* takes the same approach but requires that the first variable selected has a smaller index than a pivot element. This pivot element is the variable with most nodes in the BDD. Accordingly the second selected variable has to have a larger index than the pivot element.

2.2.2 Clause orderings

Given a CNF as input formula for the BDD compilation, the order in which clauses are added to the BDD is crucial. Consider a formula $\psi \wedge x \wedge \neg x$ where ψ is an arbitrary satisfiable CNF. If first all clauses of ψ are added to the BDD, the resulting BDD can be of large size

(depending on ψ). If then at the end x and $\neg x$ are added, the formula turns unsatisfiable and the BDD degenerates to the 0-leaf. If on the other hand x and $\neg x$ are added to the BDD as first two clauses, all other clauses will have no more impact on the BDD. Obviously the second approach would perform much better in this case for a large ψ. In general we can not determine such a 'good' clause ordering but in our application we have specific structure knowledge which we can utilize. Since clauses in our application stem from various constraints, we will refer to this also as *constraint ordering*.

3 TEST CASES

As a testbed we used product configuration formulas for a series of cars of a major German car manufacturer. The series consists of 25 different car models, each with about 300 customer-selectable options in $\mathcal{O}$ and between 300 and 400 constraints in Ψ. Looking at the distinction in section 2.1.2 we have the following numbers:

- between 20 and 30 unit constraints in Ψ_U
- between 40 and 60 cardinality constrains in Ψ_{CC}
- about 300 dependencies in Ψ_D

The corresponding CNF translations of these formulas range between 200 and 350 variables and 500 and 3000 clauses.

We distinguish two different flavors of formulas: the first set represents a restriction of the formula to technical aspects, meaning only options are considered, which are really choosable by the customer; the second set models the full configuration space including some steering codes in the set of options which are used to guide certain processes during the manufacturing of the car. Table 1 presents an overview over all instances.

Table 1. Automotive product configuration instances

	technical		full configuration space	
	# variables	# clauses	# variables	# clauses
IA1	270	979	352	2796
IA2	262	895	344	2712
IA3	268	942	350	2759
IA4	262	898	344	2715
IB1	242	704	322	2519
IB2	236	667	316	2482
IC1	251	768	331	2583
IC2	242	704	322	2519
IC3	220	594	240	638
IC4	267	952	349	2769
IC5	257	853	339	2670
ID1	246	760	325	2575
ID2	237	696	317	2511
ID3	216	597	236	641
ID4	246	765	326	2580
ID5	238	669	318	2514
ID6	216	597	236	641
IE1	240	700	319	2514
IE2	236	662	315	2476
IE3	247	745	327	2560
IE4	241	695	321	2510
IE5	246	736	326	2551
IE6	241	697	321	2512
IE7	267	946	349	2763
IE8	257	859	339	2676

4 RESULTS

A framework for automated testing and evaluating of both static and dynamic variable orderings has been implemented. We used CUDD [2] as a foundation of our implementation. The framework can handle Dimacs CNF files and produces BDDs with different reordering heuristics and also a static variable ordering. It then evaluates the resulting BDDs wrt. compilation time and BDD size and generates comparison graphs for the different heuristics.

For the static variable ordering we solved the formula with a state-of-the-art SAT solver and used its assignment stack as ordering.

4.1 The impact of the constraint ordering

As mentioned in section 2.2.2 the clause/constraint ordering can play an important role in the compilation phase. In the first tests most of our instances could not be compiled into BDDs with an arbitrary constraint ordering. We can observe this effect in the first diagram of figure 2 for a test instance (IB2). Without structuring the constraints, we reached > 50 million internal nodes after adding 100 clauses. Due to memory restrictions (4 GB) we could not add more than 200 clauses.

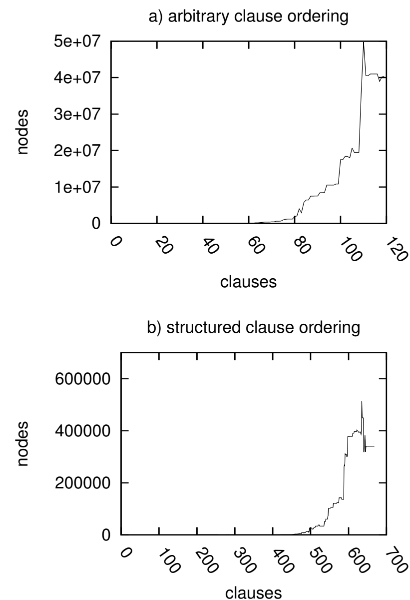

Figure 2. Impact of clause orderings (IB2)

To bypass this problem, we grouped our set of constraints according to section 2.1.2 in Ψ_U, Ψ_{CC} and Ψ_D and used the constraint ordering $\Psi_U < \Psi_{CC} < \Psi_D$, meaning first we add all the unit constraints, then we add all the cardinality constraints, and at last we add the dependencies. We will now take a closer look at the resulting BDD.

The conjoin of all the constraints in Ψ_U represents exactly one satisfying assignment. Therefore the resulting BDD is—independent of the variable ordering—a chain of n nodes for n constraints in Ψ_U. For each variable representing a necessary option, the negative edge goes to the 0-leaf and the positive edge goes to the next variable, and

[2] ftp://vlsi.colorado.edu/pub/cudd-2.5.0.tar.gz

vice versa for each variable representing a forbidden option. Figure 3 illustrates the BDD after adding all unit constraints for necessary options $n_1, \dots, n_k$ and forbidden options $f_1, \dots, f_l$.

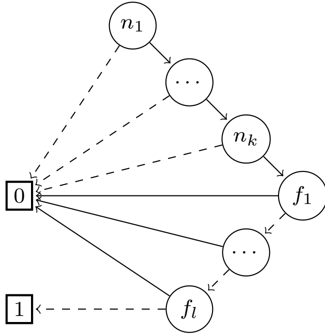

Figure 3. BDD after adding all unit constraints

Next we add all the cardinality constraints. In our context we only have to deal with 'exactly one' and 'at most one' constraints. The propositional translation of 'at most one option of $x_1, \dots, x_n$ is chosen' is

$$\bigwedge_{i \in \{1,\dots,n\}} \bigwedge_{j \in \{i+1,\dots,n\}} (\neg x_i \vee \neg x_j). \quad (1)$$

For the encoding of 'exactly one variable of $x_1, \dots, x_n$ is chosen' we simply conjoin $\left(\bigvee_{i \in \{1,\dots,n\}} x_i\right)$ to (1). The resulting BDD has (independent of the variable ordering) $2n - 1$ nodes for an 'exactly one of n' constraint and $2n - 2$ nodes for an 'at most one of n' constraint. Figure 4 illustrates such a BDD for an 'exactly one of $x_1, \dots, x_n$' constraint. In our application domain all constraints in Ψ_{CC} have disjoint variable sets (an option can only belong to one option-family). Therefore compiling all cardinality constraints to a BDD results in a chain of sub-BDDs as represented in figure 4. If one of the options in a cardinality constraint was also present in the unit constraints, the reduction property of the BDD guarantees immediate simplification. After adding the cardinality constraints and the unit constraints, the BDD size is still linear in the number of unit constraints and cardinality constraints and their respective variables.

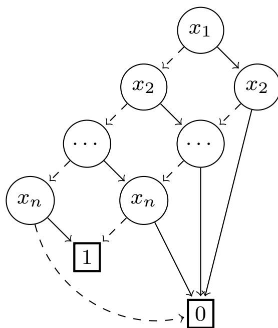

Figure 4. BDD for an 'exactly one' constraint

As a last step, the dependencies between the options Ψ_D are conjoined to the BDD. This step can enlarge the BDD significantly (exponential size in the worst-case). But our experiments show, that the knowledge already present due to the translation of the unit constraints and the cardinality constraints helps to a great extent to simplify the remaining constraints.

In the second diagram of figure 2 we can observe this effect: Adding the clauses representing unit and cardinality constraints (the first 500 clauses) goes smoothly and the resulting BDD is very small. First on adding the dependencies, the BDD size grows faster. But taking into account that we could not compile over 200 clauses with an arbitrary constraint ordering, this is a large improvement. With the help of this new constraint ordering, we were able for the first time, to compile all our industrial instances into BDDs with under two minutes per instance (most of them taking only a few seconds to compile).

4.2 Comparison of the reordering heuristics

We compared all reordering heuristics wrt. compilation time (execution time in user mode) and BDD size (total number of nodes). Our test system was a 64-Bit Linux running on an AMD Athlon 64 X2 Dual Core 4600+ with 4 GB of RAM. For each instance all 16 heuristics were tested. The results are denoted as follows:

- *var:* static variable ordering (assignment stack of SAT solver)
- *none:* ascending variable order $x_1 \dots x_n$
- *sifting:* basic sifting algorithm
- *symsift:* symmetric sifting
- *gsift:* group sifting
- *windowX:* window permutation with window size X
- *random:* random selection algorithm
- *rpivot:* random selection with pivot element

A '-c' identifies the convergent variant of a heuristics, which means it is applied until no further improvement can be observed.

Figure 5 presents an evaluation for one test instance as automatically produced by our tool. Here you can observe a typical pattern we identified: the static variable ordering often has a short compilation time, but produces large BDDs. The windowing algorithms perform better than the sifting-based algorithms in most cases. The sifting algorithms yield by far the smallest BDDs.

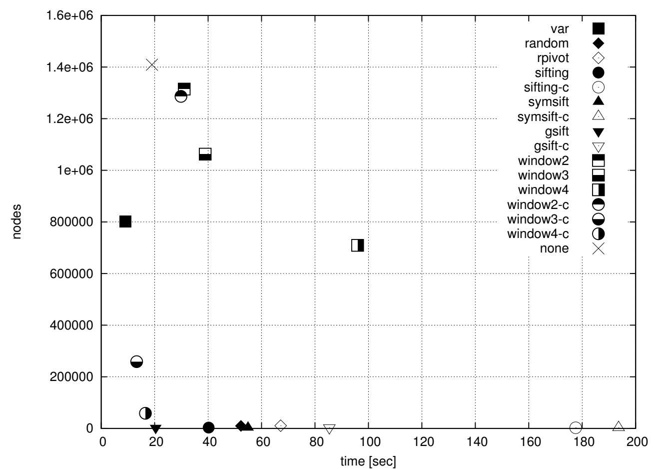

Figure 5. Heuristics comparison for ID5 (full configuration space)

Table 2 presents an overview what algorithm yielded the smallest BDD size for each test instance and what algorithm had the best compilation time. Times are noted in seconds. The table presents only instances of the full configuration space. The instances reduced to technical aspects showed similar results and are omitted here.

Table 2. Comparison of reordering algorithms (full configuration space)

	nodes	winner	time (in s)	winner
IA1	6133	gsift-c	36.41	none
IA2	3837	symsift-c	62.2	symsift
IA3	1974	symsift-c	28.3	window3-c
IA4	12016	gsift-c	48.61	window3-c
IB1	1820	gsift-c	3.37	var
IB2	2878	gsift-c	1.03	var
IC1	2539	symsift	7.6	window3-c
IC2	1411	gsift-c	8.17	var
IC3	844	symsift	0.99	var
IC4	4229	symsift-c	45.46	window3-c
IC5	2883	symsift-c	33.35	gsift
ID1	1781	symsift-c	18.18	window3-c
ID2	2702	gsift-c	10.87	var
ID3	1345	symsift-c	0.58	var
ID4	1343	symsift-c	3.54	var
ID5	2407	gsift	9.01	var
ID6	1345	symsift-c	0.6	var
IE1	1165	gsift-c	2.59	var
IE2	1313	symsift	0.93	var
IE3	3587	sifting-c	13.15	window3-c
IE4	2029	gsift	3.08	var
IE5	1853	symsift-c	10.89	var
IE6	1898	symsift-c	6.15	var
IE7	2308	symsift-c	37.91	window3-c
IE8	2233	gsift-c	34.66	window3-c

These results are summarized in table 3[3]. Here we show how many times (out of 50 instances—full and technical configuration space) each algorithm yielded the best result wrt. to size and time respectively. The aforementioned observations are justified: a static variable ordering or reordering algorithms based on windowing (especially with window size 3) have often the best compilation times at the expense of large BDD sizes. The various reorderings based on sifting, especially the convergent symmetric sifting variant, produce the smallest BDDs. Since performance in the offline phase is not too critical in knowledge compilation, sifting seems to be a viable choice for a reordering heuristics for our test instances in order to compile small BDDs with good query times.

5 CONCLUSION

In this paper we introduced a new constraint ordering for BDD compilation of industrial configuration instances. This constraint ordering uses structure-specific knowledge of the constraints at hand in order to optimize compilation time. With this ordering we were able for the first time to compile all configuration instances of our testbed—product configuration data of a major German car manufacturer—into BDDs. Most of these BDDs could be compiled in a few seconds and have surprisingly small representations (850 - 12.000 nodes). These results look very promising. There are some interesting questions like counting all constructible variants of a single car [9] or enumerating a certain number of cars with special features, that could be solved efficiently once we have a BDD representation.

[3] Complete benchmark results can be found at `http://www-sr.informatik.uni-tuebingen.de/research/confws2012-results.pdf`

Table 3. Summary of the winning heuristics

	# smallest size	# best time
var	0	26
random	0	0
randompivot	0	0
sifting	4	1
sifting-c	2	0
symsift	11	3
symsift-c	21	0
gsift	2	2
gsift-c	10	0
window2	0	0
window3	0	0
window4	0	0
window2-c	0	0
window3-c	0	15
window4-c	0	1
none	0	2

REFERENCES

[1] Jean Marc Astesana, Yves Bossu, Laurent Cosserat, and Helene Fargier, 'Constraint-based modeling and exploitation of a vehicle range at renault's: Requirement analysis and complexity study', in *Proceedings of the 13th Workshop on Configuration*, pp. 33–39, (2010).

[2] Beate Bollig and Ingo Wegener, 'Improving the variable ordering of OBDDs is NP-complete', *IEEE Transactions on Computers*, **45**(9), 993–1002, (1996).

[3] Randal E. Bryant, 'Graph-based algorithms for boolean function manipulation', *IEEE Transactions on Computers*, **35**(8), 677–691, (1986).

[4] Randal E. Bryant and Miroslav N. Velev, 'Boolean satisfiability with transitivity constraints', in *Proceedings of the CAD 2000*, volume 1855 of *Lecture Notes in Computer Science*, 85–98, Springer, Berlin, Heidelberg, Germany, (2000).

[5] Adnan Darwiche, 'Decomposable negation normal form', *Journal of the ACM*, **48**(4), 608–647, (2001).

[6] Masahiro Fujita, Yusuke Matsunaga, and Takeo Kakuda, 'On variable ordering of binary decision diagrams for the application of multi-level logic synthesis', in *Proceedings of the European Conference on Design Automation*, 50–54, IEEE Computer Society, (1991).

[7] Tarik Hadzic, Subbarayan Sathiamoorthy, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard, 'Fast backtrack free product configuration using precompiled solution space representations', in *Proceedings of the PETO 2004*, (2004).

[8] Nagisa Ishiura, Hiroshi Sawada, and Shuzo Yajima, 'Minimization of binary decision diagrams based on exchanges of variables', in *Proceedings of the ICCAD 1991*, 472–475, IEEE Computer Society, (1991).

[9] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin, 'Model counting in product configuration', in *Proceedings of LoCoCo 2010*, volume 29, 44–53, EPTCS, (2010).

[10] Wolfgang Küchlin and Carsten Sinz, 'Proving consistency assertions for automotive product data management', *Journal of Automated Reasoning*, **24**(1-2), 145–163, (2000).

[11] Nina Narodytska and Toby Walsh, 'Constraint and variable ordering heuristics for compiling configuration problems', in *Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI'07*, 149–154, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, (2007).

[12] Shipra Panda and Fabio Somenzi, 'Who are the variables in your neighbourhood', in *Proceedings of the ICCAD 1995*, 74–77, IEEE Computuer Society, (1995).

[13] Shipra Panda, Fabio Somenzi, and Barbard F Plessier, 'Symmetry detection and dynamic variable ordering of decision diagrams', in *Proceedings of the ICCAD 1994*, 628–631, IEEE Computer Society, (1994).

[14] Richard Rudell, 'Dynamic variable ordering for ordered binary decision diagrams', in *Proceedings of the ICCAD 1993*, 42–47, IEEE Computer Society, (1993).

[15] Daniel Sabin and Rainer Weigel, 'Product configuration frameworks-a survey', *IEEE Intelligent Systems*, **13**(4), 42–49, (1998).