Enumeration of valid partial configurations

Alexey Voronov, Knut Åkesson, Fredrik Ekstedt
Chalmers University of Technology, Göteborg, Sweden
{voronov, knut}@chalmers.se
Fraunhofer-Chalmers Research Centre for Industrial Mathematics, Göteborg, Sweden
fredrik.ekstedt@fcc.chalmers.se

Abstract

Models of configurable products can have hundreds of variables and thousands of configuration constraints. A product engineer usually has a limited responsibility area, and thus is interested in only a small subset of the variables that are relevant to the responsibility area. It is important for the engineer to have an overview of possible products with respect to the responsibility area, with all irrelevant information omitted. Configurations with some variables omitted we will call *partial configurations*, and we will call a partial configuration *valid* if it can be extended to a complete configuration satisfying all configuration constraints. In this paper we consider exact ways to compute valid partial configurations: we present two new algorithms based on Boolean satisfiability solvers, as well as ways to use knowledge compilation methods (Binary Decision Diagrams and Decomposable Negation Normal Form) to compute valid partial configurations. We also show that the proposed methods are feasible on configuration data from two automotive companies.

1 Introduction

Within the automotive industry it is common to have a few general platforms where each platform are highly configurable to adapt to the needs on different markets but also to satisfy the needs of individual customers. Having highly configurable products put high demands on the engineering systems to support the engineers during the development of the platforms. While having a configurable product or platform it is inevitable to also have constraints that specify what can be allowed together and what is not allowed together. These constraints can, in many situations, be defined as a set of Boolean formulas defined over finite domain variables.

Development of complex products—like in the automotive industry—is done in large teams, where each engineer is working with only a limited part of the product. For the individual engineer only a small subset of the variables from those that describe the full product, are of immediate interest. However, an important problem for the engineer is to know which combinations of variable assignments for a subset of the variables might result in a product that satisfies all constraints, that is to know *valid partial configurations*. This is important because it helps the engineer to develop solutions only for those combinations that can actually be built and sold. Overestimation of these solutions will lead to engineer doing unnecessary designs. Underestimation can lead to costly delays if an overlooked configuration is requested by a customer afterwards.

Computing the exact set of valid partial configurations is generally hard. Simply taking configurations of the complete products and projecting them on relevant variables is not feasible, as practical problems can have 10^{120} and more buildable complete products.

One of the ways to get exact set of valid partial configurations is to existentially quantify all irrelevant variables from the formula that represents the conjunction of all configuration constraints, for example using resolution inference rule [Robinson, 1965; Davis and Putnam, 1960], and then use a standard algorithm to enumerate all (complete) assignments of the new simplified formula. A simple enumeration of complete assignments searches for a satisfying assignment, adds it to the result, and also adds it to the current set of constraints as a blocking constraint, forbidding future search from returning it again. The disadvantage of this approach is that the formula size can grow significantly after quantification. In this paper we present a modification of this enumeration algorithm that does not require existential quantification of variables to enumerate partial configurations (Section 4.1). Similar algorithm can be found in [Gebser *et al.*, 2009].

The problem of partial configurations can also be tackled by the widely used interactive configurators. In an interactive configurator a user selects values for variables one by one. The configurator should guide the user so that at any point there exist at least one way to complete the configuration without changing any of the earlier decisions, in this case a configurator is *backtrack-free*. Configurator should also be *complete* meaning that if a configuration is allowed according to the constraints, configurator should allow it. Having such complete and backtrack-free configurator, it is possible to automatically check all (partial) assignments of values to the relevant variables, and the configurator will show which of them are valid. If a configurator is not backtrack-free, it can overestimate allowed partial configurations. If it is not complete, it will underestimate them.

Previous work on methods for building interactive configurators started as extensions of Constraint Satisfaction Problem (CSP) with conditional and dynamic formulations [Dechter and Dechter, 1988; Mittal and Falkenhainer, 1990; Soininen and Gelle, 1999; Sabin and Freuder, 1998; Gottlob *et al.*, 2007]. However, supported implementations of these algorithms are not readily available. Binary Decision Diagrams (BDDs) [Bryant, 1986] is a knowledge compilation method [Darwiche and Marquis, 2002] successfully used for configuration [Hadzic *et al.*, 2004], especially for real-time interactive configuration. However, BDDs suffer from memory explosion for many datasets of practical size. Other knowledge compilation methods used for configuration include automata representation [Amilhastre *et al.*, 2002], Tree-of-BDDs [Subbarayan, 2005], Joint Matched CSP [Subbarayan and Andersen, 2005], Decomposable Negation Normal Form (DNNF) [Darwiche and Marquis, 2002] (especially Deterministic DNNF for model counting [Kübler *et al.*, 2010]), as well as combinations of search and BDDs [Norgaard *et al.*, 2009]. Recently, Boolean Satisfiability Solvers (SAT-solvers) emerged as an alternative to work with configurations [Sinz *et al.*, 2003; Küchlin and Sinz, 2000; Sinz *et al.*, 2006; Janota, 2008], including interactive configurators [Janota, 2010].

In this paper we show how a SAT-solver can be used to enumerate partial configurations based on modification of standard enumeration algorithm, and based on checking every partial assignment inspired by interactive configuration. We also show how existing DNNF algorithms can be used to enumerate partial configurations. We show feasibility of a SAT-solver based implementation on configuration data from two automotive companies.

The paper is organized as following. Section 2 covers formal preliminaries, Section 3 gives a motivating example, Section 4 presents the algorithms. Section 5 provides experimental results and discussion, and Section 6 concludes the paper.

2 Preliminaries

The configuration problem is a triple $\mathcal{P} = \langle X, D, C \rangle$, where $X = \{x_1, x_2, \ldots, x_K\}$ is a set of variables, $D = \{D_1, D_2, \ldots, D_K\}$ is a set of corresponding finite domains, and $C = \{C_1, C_2, \ldots, C_J\}$ is a set of propositional formulas over atomic propositions $x_k = v$ where $v \in D_k$, specifying conditions that the variable assignments have to satisfy.

A *complete assignment* to a configuration problem $\mathcal{P}$ is a function $A : X \to D$ which is defined for all $x_k \in X$. A *valid* complete assignment (or solution) to $\mathcal{P}$ is a complete assignment A for which each C_j is satisfied. A *partial* assignment to $\mathcal{P}$ is a partial function $B : X \to D$ defined for variables $x_k \in Y \subseteq X$. We will write *vars*$(B) = Y \subseteq X$ to denote the set of variables of B, or the *scope* of B. We will call a partial assignment *valid* iff it can be extended to a valid complete assignment. We will use $\mathcal{P}[B]$ to denote the simplified problem obtained by setting the variables defined in B.

3 Motivating example

Configuration problems describing complete products can have thousands of variables and hundreds of thousands constraints. An engineer, or a group of engineers, is usually responsible only for a subset of the variables. It could be that it is a requirement to design all possible solutions within the responsibility area, in case someone will order such a product. In such a case it can be expensive to have overestimated set of valid configurations, since engineers will spend time designing forbidden ones. Underestimations are also bad, since they lead to delays for designing a solution for ordered, but missed configuration.

The problem can be illustrated by the following example of a simple car configuration. Let $X = \{body, engine, transmission\}$ be the set of variables, and $D = \{\{mini, sedan, suv\}, \{gasoline, diesel, electric\}, \{manual, auto, evt\}\}$ be the set of corresponding domains. Let the following be the set of constraints C:

- $\neg((body = mini) \land (engine = gasoline))$
- $\neg((body = mini) \land (engine = diesel))$
- $\neg((body = sedan) \land (engine = electric))$
- $\neg((body = suv) \land (engine = gasoline))$
- $(engine = electric) \to (transmission = evt)$
- $(transmission = evt) \to (engine = electric)$

Valid assignments of $\mathcal{P} = \langle X, D, C \rangle$ can be presented, for example, in a tabular form, as shown in Table 1. Each row in the table corresponds to an assignment, and each column corresponds to a variable. Each cell contains a value assigned to the corresponding variable.

Let us suppose that there is a group of engineers that are interested only in connection between *body* and *transmission*, and they would like to disregard all information about *engine*. So they define the limited scope to be $S = \{body, transmission\}$. Valid partial assignments for S are presented in Table 2.

One way to get the partial assignments is to enumerate all complete assignments, project them onto the relevant variables (remove the irrelevant columns from the table), and remove duplicate partial assignments (rows). This approach is infeasible in practice, as some industrial examples from automotive industry have 10^{120} allowed complete assignments. However, scopes of interest for the engineers may have less

Table 1: Valid complete assignments

body	engine	transmission
mini	electric	evt
sedan	gasoline	manual
sedan	gasoline	automatic
sedan	diesel	manual
sedan	diesel	automatic
suv	diesel	manual
suv	diesel	automatic
suv	electric	evt

Table 2: Valid partial assignments

body	transmission
mini	evt
sedan	manual
sedan	automatic
suv	manual
suv	automatic
suv	evt

than a thousand valid partial assignments, which is computable using methods presented below. Approximations of this approach are found in practice, where instead of all valid assignments, only the ones that correspond to the products built during the last year (for example) are considered, which gives an underestimation of the answer. Clearly, there is a need for a better method.

4 Enumerating valid partial assignments

This sections presents two algorithms adopted to solve the problem based on *satisfiability solvers*. By a satisfiability solver we mean a tool that is able to answer whether an instance of a configuration problem has at least one valid complete assignment, and provides one if such exists. For example, tools for solving Constraint Satisfaction Problems (CSP-solvers) and Boolean Satisfiability Problems (SAT-solvers) can be used for this purpose. This section also shows how DNNF algorithms can be used to enumerate valid partial configurations.

4.1 Searching for complete, then forbidding partial

One way to enumerate all valid complete assignments is to iteratively search for any valid complete assignment, and in addition to adding it to the result, add a negation of it as a blocking constraint to the existing set of constraints. This algorithm can be modified to enumerate valid partial assignments, as shown in Algorithm 1. When a solver returns the first complete assignment, the assignment is projected onto the relevant scope. This partial assignment is returned as the first element of the result, and also added as a blocking constraint, ensuring that the solver will not return any (complete) assignment that will contain the partial one. Then this process is repeated.

The ability of the solver to incrementally add blocking constraints, while still keeping previously inferred information, is very important for the good performance of this algorithm. This is supported by, for example, Minisat-like solvers [Een and Sörensson, 2004; Een and Sörensson, 2003].

4.2 Enumerating partial, then extending

In this approach it is necessary to enumerate *all* partial assignments, and try to extend each of them to a valid complete assignment using a solver. Just checking a partial assignment against each of the constraints in isolation is not enough, because there could be dependencies between variables that are not visible within the local scope, but are only visible within

Algorithm 1 Search for complete, then forbid partial

```
input: problem $\mathcal{P} = \langle X, D, C \rangle$, relevant variables $S \subseteq X$
$C' \leftarrow C$
$\mathcal{P}' \leftarrow \mathcal{P}$
result ← {}
while sat($\mathcal{P}$): /* ask solver */
    $A \leftarrow$ assignment($\mathcal{P}$) /* assignment from solver */
    $B \leftarrow A$ projected on $S$
    result ← result $\cup B$
    $C' \leftarrow C' \wedge \neg(B$ as constraint)
    $\mathcal{P}' \leftarrow \langle X, D, C' \rangle\rangle$
return result
```

the complete scope. The solver can be used as following: each partial assignment is added as an extra constraint to the set of configuration constraints, and removed after the solver has returned an answer. The key to the good performance in this method is in the ability of the solver to cheaply add and retract constraints consisting of atomic propositions; again, Minisat-like solvers have this feature.

Algorithm 2 Enumerate partial, then extend

```
input: problem $\mathcal{P} = \langle X, D, C \rangle$, relevant variables $S \subseteq X$
output: valid partial assignments
result ← {}
for $B$ in allAssignments($S$):
    if sat( $\mathcal{P}[B]$ ):
        result ← result $\cup B$
return result
```

An example illustrating this method is presented in Table 3. The columns for *body* and *transmission* contain all possible (not only valid) partial assignments for S. The table must be extended with the columns for variables $(X \setminus S)$, in this case it is only one, *engine*. If there is at least one valid complete assignment that contains the partial one for the row, the values for all variables are written in the row. Otherwise, a "—" indicates that there is no such valid complete assignment, and the partial assignment is not valid.

Table 3: Illustration of Algorithm 2 (Enumerate partial, then extend).

body	transmission	*engine*
mini	manual	—
mini	autmatic	—
mini	evt	*electric*
sedan	manual	*gasoline*
sedan	automatic	*gasoline*
sedan	evt	—
suv	manual	*diesel*
suv	automatic	*diesel*
suv	evt	*electric*

An advantage of Algorithm 1 compared to Algorithm 2 is that it builds upon heuristics of an underlying solver to skip checking many of the non-allowed assignments. A disadvantage is that to find the next partial assignment, it is necessary to process a larger (increased by one) set of constraints; this could be a problem when it is necessary to produce millions of partial assignments. But when there is a small number of valid partial assignments, or a user specifically asked for the first hundred of assignments as a sample, and there are many non-allowed configurations, Algorithm 1 can be beneficial.

4.3 Knowledge compilation: DNNF

Knowledge compilation is a family of approaches that addresses intractability of many Artificial Intelligence problems. A propositional model is compiled in an off-line phase in order to support some queries in polytime [Darwiche and Marquis, 2002]. Binary Decision Diagrams (BDDs) [Bryant, 1986] belong to this family and received substantial attention as a tool for configuration problems [Hadzic *et al.*, 2004]. Decomposable Negation Normal Form (DNNF) [Darwiche, 2001] is a data structure used in knowledge compilation for which BDD is a special case. It is more succinct than BDDs and its compilation time is often shorter than that of BDDs [Subbarayan *et al.*, 2007]. DNNF supports smaller number of tractable operations than BDD, while still allowing polytime existential quantification (forgetting) and assignments enumeration, which together allow polytime partial assignments enumeration once DNNF is compiled.

Formally, a propositional formula a is in negation normal form (NNF) if and only if a is either a positive or negative atomic proposition (a *literal*); a conjunction $\wedge_i a_i$; or a disjunction $\vee_i a_i$ where each a_i is in negation normal form. A formula in NNF f is *decomposable* (DNNF) if and only if for any conjunction $a = a_1 \wedge \cdots \wedge a_n$ no atomic propositions are shared by any conjuncts in a: $\text{ATOMS}(a_i) \cap \text{ATOMS}(a_j) = \emptyset$ for every $i \neq j$. A formula in NNF is *smooth* if for every disjunction $a = a_1 \vee \cdots \vee a_n$, $\text{ATOMS}(a) = \text{ATOMS}(a_i)$ for every i.

Existential quantification of variables from DNNF is presented in Algorithm 3 [Darwiche, 2001]. Every occurence of irrelevant variable is replaced by *true*, and the formula is simplified accordingly. This procedure preserves decomposability, and its running time is linear in the DNNF size.

Algorithm 3 FORGET – existential quantification on DNNF

```
input: relevant variables $S \subseteq X$, DNNF $f$
output: DNNF with variables $X \setminus S$ existentially quantified
if $f$ is a Literal $l$:
    if $\text{VAR}(l) \in S$:
        return $f$
    else
        return true
else if $f$ is a conjunction $a_1 \wedge \cdots \wedge a_n$:
    return $\text{FORGET}(a_1) \wedge \ldots \wedge \text{FORGET}(a_n)$
else if $f$ is a disjunction $a_1 \vee \cdots \vee a_n$:
    return $\text{FORGET}(a_1) \vee \ldots \vee \text{FORGET}(a_n)$
```

Enumeration of assignments of DNNFs is shown in Algorithm 4 [Darwiche, 2000], where each assignment is represented as a set of literals, and $\times$ is a Cartesian product on them:

$$\{N_1, \ldots, N_n\} \times \{M_1, \ldots, M_m\} = \{N_1 \cup M_1, \ldots, N_n \cup M_m\}.$$

The complexity of enumerating the models of a smooth DNNF f is $O(mn^2)$, where m is the size of f and $n = |\text{MODELS}(f)|$ [Darwiche, 1998].

Algorithm 4 MODELS – enumerating assignments of DNNF

```
input: smooth DNNF $f$
output: (complete) valid assignments of $f$, as sets of literals
if $f$ is a Literal $l$:
    return $\{\{l\}\}$;
else if $f$ is a conjunction $a_1 \wedge \cdots \wedge a_n$:
    return $\text{MODELS}(a_1) \times \cdots \times \text{MODELS}(a_n)$;
else if $f$ is a disjunction $a_1 \vee \cdots \vee a_n$:
    return $\text{MODELS}(a_1) \cup \cdots \cup \text{MODELS}(a_n)$.
```

The overall process of using DNNF is shown in Algorithm 5. DNNF for the car example is shown on Figure 1a. DNNF with variable *engine* forgotten is shown on Figure 1b, and its valid partial assignments can be found in Table 2.

Algorithm 5 Knowledge compilation based approach

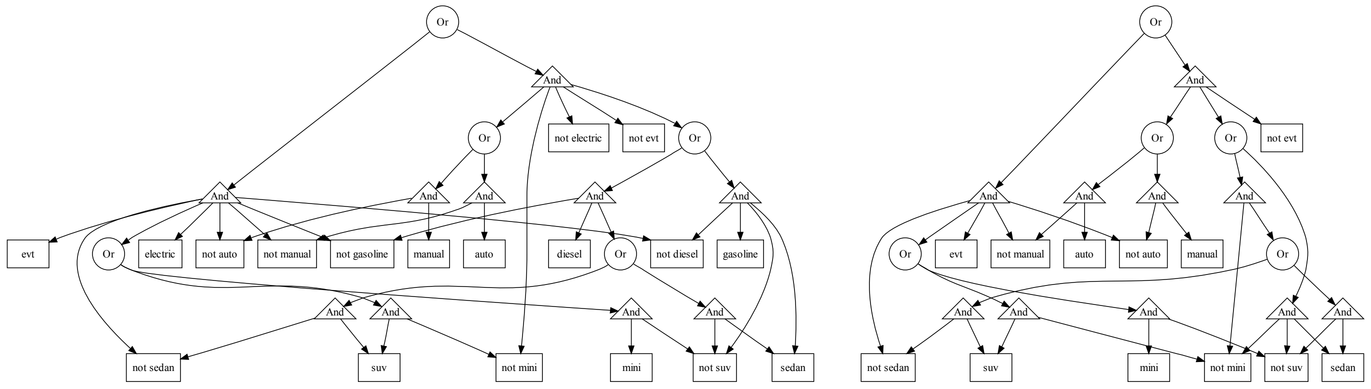
```
input: problem $\mathcal{P} = \langle X, D, C \rangle$, relevant variables $S \subseteq X$
output: valid partial assignments
$f_1 \leftarrow \text{COMPILE}(\mathcal{P})$
$f_2 \leftarrow \text{FORGET}(S, f_1)$ /* see Algorithm 3 $*$ /
$m \leftarrow \text{MODELS}(f_2)$ /* see Algorithm 4 $*$ /
return $m$
```

Polynomial time guarantee for assignments enumeration operation is an advantage of DNNF. However, the compilation time of arbitrary constraints into DNNF is in general exponential. When the data changes rarely, this time is amortized among multiple queries, but when the data changes very often, this off-line stage does not pays off.

DNNF have an important advantage: if it is smooth and *deterministic*, it can be used to count the assignments [Darwiche, 2000]. An NNF formula a is *deterministic* if for every disjunction $a = a_1 \vee \cdots \vee a_n$, every pair of disjuncts in a is logically inconsistent; that is, $a_i \wedge a_j \models false$ for every $i \neq j$. Unfortunately, operation FORGET does not preserve determinism, and counting in such case will give overestimated answer. However, even overestimated answer can be useful in some cases. It is also possible to recompile the resulting DNNF again into a deterministic one. Some practical applications of counting for configuration using DNNF were considered in [Kübler *et al.*, 2010].

5 Experimental results

We analyzed the data from two automotive companies: three datasets from the first company, and one dataset from the sec-



(a) Full DNNF. (b) DNNF with variable *engine* existentially quantified.

Figure 1: DNNFs for the car example.

ond, denoted as A, B, C and D, respectively. Details about datasets are presented in Table 4.

We implemented Algorithms 2 and 1 on top of Sat4j solver [Le Berre and Parrain, 2010].

A knowledge compilation tool was developed based on BDD package BuDDy [Lind-Nielsen, 2002]. The pre-ordering algorithms from [Narodytska and Walsh, 2007] were implemented for sorting variables and restrictions, using inflation parameter $r = 1.5$ in the clustering step. A simplified version of the MCL clustering algorithm [van Dongen, 2000] was used, skipping the truncation heuristics and the sparse matrix multiplication tools. No post-ordering of the variables was included. The tool was able to handle only the smallest dataset.

Another attempt to use knowledge compilation involved c2d compiler [Darwiche, 2004] that compiles propositional formulas to deterministic DNNF. Algorithms 3 (FORGET) and 4 (MODELS) were implemented to work with the DNNF output of c2d. Using another DNNF compiler sharpSAT [Muise *et al.*, 2010] resulted in segmentation faults on some of the datasets, and its debugging is underway.

Sat4j and c2d require the input to be in Conjunctive Normal Form (CNF), that is it have to be a conjunction of clauses, and each clause is a disjunction of literals. Each literal is either a positive or negative atomic proposition. Constraints were converted to CNF using Tseitin encoding [Tseitin, 1968].

Two times were measured. The first time measured was preprocessing or off-line time. This included, for example, DNNF compilation, and initial constraint propagation. The results are presented in Table 5. BDD-based implementation was not able to complete the compilation process of larger instances. c2d compiler ran out of 2 GB memory limit (it is available only as a 32 bit application) compiling the largest dataset A.

The second time measured was the on-line time, or the time to actually compute the valid partial configurations for one given scope, while utilizing results from the off-line phase. The results are presented in Table 6. SAT-based solution is very robust on the datasets, even without having theoretical guarantees on running times. The BDD-based solution, when

Table 4: Problem properties.

	A	B	C	D
Variables	511	446	92	217
Domain size, average	6.3	2.4	6.1	3.3
Domain size, max	108	82	75	59
# of assignments	10^{150}	10^{87}	10^{55}	10^{85}
# of valid assignments	10^{124}	10^{57}	10^{49}	10^{33}
CNF clauses	65183	1121	341	9010
DNNF nodes	n/a	5071	5009	528583
Partial scope, variables	6	17	3	8
# of valid partial assignments	200	13770	25	382

the BDD was successfully built, is the fastest. The reason why DNNF-based method appears to be slow could be a non-optimal implementation of Algorithms 3 and 4.

6 Conclusions

In this paper we looked at the problem of computing allowed partial combinations, which is important for engineers working with product development. We presented several algorithms, two of which are suitable for SAT-solvers, and one that is based on DNNF. Our experiments showed that SAT-based implementation can handle large datasets from automotive industry quite efficiently. Preliminary experiments with knowledge compilation tools showed that available DNNF compilers cannot handle the largest dataset within the memory and time limits. However, DNNF-based method has

Table 5: Time for compilation/initial clause learning, seconds.

	A	B	C	D
Sat4j, Alg 1	2	2	1	2
BuDDy	timeout	timeout	40	timeout
c2d	out-of-mem	240	2	20

Table 6: Time to compute valid partial assignments (FORGET+MODELS), seconds.

	A	B	C	D
Sat4j, Alg 1	10	29	0.12	3
BuDDy	n/a	n/a	0.01	n/a
DNNF from c2d	n/a	681*	0.15*	22*

*Based on own, unoptimized implementation.

the advantages of polynomial time guarantee on the on-line phase, and the ability to count the assignments when DNNF is determenistic.

Acknowledgements

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

[Amilhastre *et al.*, 2002] Jérôme Amilhastre, Helene Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs — Application to configuration. *Artificial Intelligence*, 135:199–234, 2002.

[Bryant, 1986] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17(1):229–264, 2002.

[Darwiche, 1998] Adnan Darwiche. Model-Based Diagnosis using Structured System Descriptions. *Journal of Arti cial Intelligence Research*, 8:165–222, 1998.

[Darwiche, 2000] Adnan Darwiche. On the tractable counting of theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics*, 2000.

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

[Darwiche, 2004] Adnan Darwiche. New Advances in Compiling CNF to Decomposable Negation Normal Form. In *ECAI 2004*, 2004.

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[Dechter and Dechter, 1988] Rina Dechter and Avi Dechter. Belief maintenance in Dynamic Constraint Networks. In *AAAI-88*, pages 37–42, 1988.

[Een and Sörensson, 2003] Niklas Een and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[Een and Sörensson, 2004] Niklas Een and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2919/2004:502–518, 2004.

[Gebser *et al.*, 2009] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Solution enumeration for projected Boolean search problems. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 71–86, 2009.

[Gottlob *et al.*, 2007] Georg Gottlob, Gianluigi Greco, and Toni Mancini. Conditional Constraint Satisfaction: Logical Foundations and Complexity. In Manuela M. Veloso, editor, *IJCAI-2007*, pages 88–93, Hyderabad, India, January 2007.

[Hadzic *et al.*, 2004] Tarik Hadzic, Sathiamoorthy Subbarayan, R.M. Jensen, Henrik Reif Andersen, J. Mø ller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. *PETO conference*, 2004.

[Janota, 2008] Mikolas Janota. Do SAT solvers make good configurators? *12th International Software Product Line Conference. First Workshop on Analyses of Software Product Lines*, pages 1–5, 2008.

[Janota, 2010] Mikolas Janota. *SAT Solving in Interactive Configuration (PhD thesis)*. PhD thesis, University College Dublin, 2010.

[Kübler *et al.*, 2010] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. Model Counting in Product Configuration. *Electronic Proceedings in Theoretical Computer Science*, 29(LoCoCo):44–53, July 2010.

[Küchlin and Sinz, 2000] Wolfgang Küchlin and Carsten Sinz. Proving Consistency Assertions for Automotive Product Data Management. *Journal of Automated Reasoning*, 24(1):145–163, February 2000.

[Le Berre and Parrain, 2010] Daniel Le Berre and Anne Parrain. The Sat4j library , release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[Lind-Nielsen, 2002] Jø rn Lind-Nielsen. BuDDy: A BDD package. http://buddy.sourceforge.net, 2002.

[Mittal and Falkenhainer, 1990] Sanjay Mittal and Brian Falkenhainer. Dynamic Constraint Satisfaction Problems. In *AAAI-90*, pages 25–32, 1990.

[Muise *et al.*, 2010] Christian Muise, Sheila McIlraith, J.Christopher Beck, and Eric Hsu. Fast d-DNNF Compilation with sharpSAT. In *AAAI 2010*, pages 54–60, 2010.

[Narodytska and Walsh, 2007] 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*, pages 149–154, Hyderabad, India, 2007. Morgan Kaufmann Publishers Inc.

[Norgaard *et al.*, 2009] Andreas Hau Norgaard, Morten Riiskjaer Boysen, Rune Moller Jensen, and Peter Tiedemann. Combining Binary Decision Diagrams and Backtracking Search for Scalable Backtrack-Free Interactive Product Configuration. In *Proceedings of the 21st International Joint Conferences on Artificial Intelligence(IJCAI-09) Workshop on Configuration*, 2009.

[Robinson, 1965] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sabin and Freuder, 1998] Mihaela Sabin and Eugene C. Freuder. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In *Proceeding of Constraint Programming (CP-98)*, 1998.

[Sinz *et al.*, 2003] 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*, 17(01):75–97, August 2003.

[Sinz *et al.*, 2006] Carsten Sinz, Wolfgang Küchlin, Dieter Feichtinger, and Georg Görtler. Checking Consistency and Completeness of On-Line Product Manuals. *Journal of Automated Reasoning*, 37(1):45–66, August 2006.

[Soininen and Gelle, 1999] Timo Soininen and Esther Gelle. Dynamic Constraint Satisfaction in Configuration. In *AAAI-99, Workshop on Configuration*, pages 95–100, 1999.

[Subbarayan and Andersen, 2005] Sathiamoorthy Subbarayan and Henrik Reif Andersen. Linear Functions for Interactive Configuration Using Join Matching and CSP Tree Decomposition. In *Configuration Workshop at IJCAI'05*, pages 7–12, 2005.

[Subbarayan *et al.*, 2007] Sathiamoorthy Subbarayan, Lucas Bordeaux, and Youssef Hamadi. Knowledge Compilation Properties of Tree-of-BDDs. In *AAAI 2007*, pages 502–507, 2007.

[Subbarayan, 2005] Sathiamoorthy Subbarayan. Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 3524/2005:351–365, 2005.

[Tseitin, 1968] Gregory S. Tseitin. On the complexity of derivation in propositional calculus. In A.O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian)*, pages 234–259. Steklov Mathematical Institute, 1968.

[van Dongen, 2000] Stijn van Dongen. A cluster algorithm for graphs. Technical Report INS-R0010., 2000.