An Experience Report on the Optimization of the Product Configuration System of Renault *

Hao Xu, Souheib Baarir, Tewfik Ziadi, Siham Essodaigui, Yves Bossu, Lom Messan Hillah

An Experience Report on the Optimization of the Product Configuration System of Renault*

Hao Xu
LIP6
France
hao.xu@lip6.fr

Souheib Baarir
EPITA
France
souheib.baarir@lip6.fr

Tewfik Ziadi
LIP6
France
tewfik.ziadi@lip6.fr

Siham Essodaigui
Renault
France
siham.essodaigui@renault.com

Yves Bossu
Renault
France
yves.bossu@renault.com

Lom Messan Hillah
Meritis
France
lom-messan.hillah@lip6.fr

***Abstract*—The problem of configuring a variability model is widespread in many different domains. Renault has developed its technology internally to model vehicle diversity. This technology relies on the approach known as knowledge compilation to explore the configurations space. However, the growing variability and complexity of the vehicles' range hardens the space representation problem and may impact performance requirements. This paper tackles these issues by exploiting symmetries that represent isomorphic parts in the configuration space. The extensive experiments we conducted on datasets from Renault show our approach's robustness and effectiveness: the achieved gain is a reduction of 52.13% in space representation and 49.81% in processing time on average.**

***Index Terms*—Knowledge compilation, Product line, Variability model, Value Symmetries**

I. Introduction

Variability Modeling Problems [1] are pervasive in real life. Such problems are crucial in the car industry, as they touch all business activities, including design by engineers, manufacturing, vehicle documentation, and marketing forecasts. As an example of Variability Model (VM) in that industry, let's consider a VM **M**: it has a variable $model$, which represents the vehicle model with values range in the domain $\{m_1, m_2\}$; a variable $fuel\_type$ with the domain $\{petrol, diesel, LPG\}$. Then, variable dependencies describe activity-related constraints (business, technical, legal requirements, and many others). For instance, the constraint $model = m_1 \Rightarrow fuel\_type = LPG$ leads to four possible combinations that form the different configurations for this vehicle. We refer to the set of possible configurations for a VM as the *configuration space* or the *solution space*.

Renault is a world-leading automobile manufacturer that uses such VMs to model its vehicle range. Some ranges of vehicles of Renault can reach 10^{32} possible configurations. For these large VMs, Renault has plenty of variants and complex business requirements for internal and external usages. For example, the inside usages cover variant designs for engineering, sales, calculation of manufacturing cost, forecast data, and many other departments. In each of these designs, it will require performing queries on populations of vehicles. These queries are always specified with several variable assignments, and the purpose of such queries could be checking the satisfiability or looking for the satisfying configurations. For the external usages, the most common one is the online configuration of vehicles that customers are entitled to: they can choose a vehicle's model and the motor type or other options. The manufacturer needs to respond to those customers' requests within less than a second, with the set of cars matching the requested configuration. A similar case of customization has been discussed in [2].

Responding to the requests of internal applications or online customer configurations boils down to searching for and returning a solution from the *configuration space* of the VMs. From the theoretical point of view, constructing the *configuration space* is equivalent to solving a typical Constraint Satisfaction Problem (CSP) [3]. Indeed, a VM can straightforwardly translate to a CSP [4], whose solutions correspond to the *configuration space*. Furthermore, answering requests can happen in two ways: the first, the natural one, is to solve a CSP problem at every request. Taking an online configuration as an example, it becomes a new set of constraints when inputting a configuration request. It is then combined with the CSP representing the VM, deriving a new CSP problem. If this latter has a solution, then the asked configuration is possible. While this solution is simple, its computational cost (time) is expensive since one has to solve an NP-complete problem at each request [5]. However, if the underlying problem falls into a polynomial category, or the VM is too easy to solve (less than a second), it can be solved many times cost-less. When evaluating the VMs of Renault , it turns out that the considered VMs are neither polynomial nor easy to solve. In particular, using state-of-the-art solvers (e.g., `choco` [6], `absCon` [7]) on the benchmarks reveals that around 30% of the VMs are solved in more than 5 seconds to get only one solution. Thus, responding to requests in this way is unacceptable.

The second possible solution is to build, once and for all, the *configuration space*, using an approach known as *knowledge compilation* [8]. The idea of *knowledge compilation* is using symbolic structures (e.g., BDD [9], SDD [10]) to

* A French automotive manufacturer

represent the problem *configuration space*. Of course, building the *configuration space* with such symbolic structures is more complicated than finding just one solution. However, since this activity can run offline and the configuration system can reuse such a compiled knowledge for all requests, this approach seems to be a good compromise. This second solution is the one Renault has retained, implemented, and used for a long time. Though, it is subject to a critical limit: the memory size of the symbolic structure. Indeed, for the internal or external usages, the symbolic structure reside in memory, and all requests operate from there. The total size of such compiled structures can reach dozens of Gigabytes, which may bring potential pressure for the memory usage. So far, several heuristics and optimizations control the pressure on the memory. Here, we want to go further and improve the system by exploring what is called *Symmetries* [11] that represent isomorphic parts in the configuration space.

To illustrate symmetries, let's consider again the aforementioned variability model **M** augmented with a new variable, *color*, ranging in domain $\{C_1, C_2, ..., C_{10}\}$ (representing ten possible colors). If all colors are interchangeable without any constraint, then the number of solutions in the initial model **M** is multiplied by ten. However, storing all these configurations in the solution space is not mandatory since we can obtain the whole set of configurations by only knowing the solutions of the initial model **M** augmented with only one value for the *color* variable. Indeed, the interchangeability property if the domain of the *color* variable is an example of symmetries.

In this context, this paper describes our approach to exploit *symmetries* to improve the configuration system of Renault. In addition, our requirement of this study is to avoid radical changes on this system, because such changes may equally lead to modifications of the related applications, which may bring potential and unnecessary risks. Thus and as we will show in the rest of the paper, our approach is integrated with the current system in a weakly coupled manner, altering it as little as possible while optimizing it. The contributions of this paper are thus:

1) A detailed explication of symmetries' effects over the symbolic structure of Renault's configuration system.
2) Revisiting the product configuration system by exploiting symmetries.
3) A validation of the proposed approach with massive experiments based on the actual industrial CSP instances that show the robustness and effectiveness of our approach.

The paper is organized as follows. Section II firstly presents the basic concepts of CSP, then briefly describes how a VM translates as a CSP, introduces the current configuration system of Renault, and details the motivation for exploiting symmetries. Section III presents symmetries in CSP and explains how they can be profited to compress the configuration space with an example. Section IV details the contribution of this paper. It first revisits the current product configuration system, then describes how we have extended the workflow of the current configuration system to integrate symmetries. Section V presents and discusses the experimental results of our method. Section VI presents the related work, while section VII concludes this paper.

II. A Constraint-Based Representation of Vehicule Diversity

This section provides several basic definitions and notations of CSPs that the remaining of the paper will use. We then introduce the Renault's variability models and their relations with CSPs. We present the current configuration system and end the section by explaining our working constraints and motivations of exploring symmetries.

A. Definitions and Notations

A CSP is a triple (X, D, C), where X is a set of variables, D is a set of domains, and C is a set of constraints. Each variable $x \in X$ is associated with a domain $Dom(x) \in D$.

A *constraint*, $c \in C$, can be intentional or extensional:

- *Extension constraint:* also called table constraint is defined by enumerating the list of tuples that are *allowed* or *forbidden*. It is of the form $extension(X, S)$ where $X = \langle x_1, .., x_n \rangle$ and S is a set of supported/forbidden value tuples, $S = \langle \langle d_1, ..., d_n \rangle, ... \rangle$ (with $d_i \in Dom(x_i)$).
- *Intension constraint:* it is a constraint of the form $intension(X, P)$ where $X = \langle x_1, ..., x_n \rangle$ is a sequence of n variables (the scope of the constraint), and P is a predicate expression, with n formal parameters, on the variables of X.

A *literal* is a statement of the form x = d, where $d \in Dom(x)$. An *assignment* is a set of literals covering all the variables in X. A *partial assignment* is a set of literals covering a subset of X. A *solution* is an assignment consistent with all the constraints in C.

B. Renault's Variability Models as CSPs

Mapping VMs to CSPs is a process, which uses CSP variables to model vehicles features (such as the $fuel\_type$, $model$...), uses variables' domains to model available choices of the features (such as the $petrol$ for $fuel\_type$, m_1 for $model$...), and finally translates the constraints under the form of $extensions$ or $intensions$. The use of one or the other form depends on the choice made during the VM modelling process. More details can be found in [12].

We give as example a CSP p generated from the VM of Renault. We will use this example for the rest of the paper:

- $X = \{model, fuel\_type, airconditioning, dustfilter\}$
- $D = \{\{m_1, m_2\}, \{petrol, diesel, LPG\}, \{manual, auto, none\}, \{with, none\}\}$
  with, $Dom(model) = \{m_1, m_2\}$,
  $Dom(fuel\_type) = \{petrol, diesel, LPG\}$,
  $Dom(airconditioning) = \{manual, auto, none\}$ and
  $Dom(dustfilter) = \{with, none\}$

We have one $extension$ constraint c_1:

$$c_1 : extension(\langle model, fuel\_type \rangle, \langle \langle m_1, LPG \rangle,$$

$$\langle m_2, Petrol\rangle, \langle m_2, diesel\rangle, \langle m_2, LPG\rangle\rangle)$$

We have several $intension$ constraints:

$$c_2 : intension(\langle model, fuel\_type, airconditioning\rangle, \langle(((model = m_1) \vee (model = m_2)) \wedge ((fuel\_type = petrol) \vee (fuel\_type = diesel))) \Rightarrow airconditioning = auto\rangle)$$

$$c_3 : intension(\langle airconditioning, dustfilter\rangle, \langle airconditioning = manual \Rightarrow dustfilter = with\rangle)$$

It's important to mention that, in our context, all manipulated variables have only finite domains, therefore any intension constraint can map to an equivalent extension constraint.

C. Overview of the current configuration system

Renault's current compilation system can be divided into two parts: offline part generates and saves the configuration space while online part deals with variant requests. This section details the offline part and highlights our motivation of exploring symmetries. In addition, we give an overview of this system and specify the challenges and our working constraints.

a) Current configuration system: as mentioned in the introduction, the configuration space is represented by a symbolic structure, which implicitly saves all the possible configurations. More specifically, it is based on a private compiled representation of vehicle diversity in the form of a cluster tree, which has been used in various applications since 1995 in Renault [13]. Here, we detail the offline process of the construction of the cluster tree (more details can be found in [13], [14]).

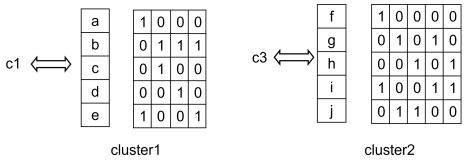

Fig. 1: $cluster_1$ and $cluster_2$.

- **Implementation of a cluster**: a cluster is a group of variables associated to a set of partial solutions of a set of constraints. These latter should only involve the variables in the cluster. The compilation process encodes literals as Boolean variables. Each partial solution is encoded as a vector of bits. For the CSP p, consider that $cluster_1$ refers variables $model$ and $fuel\_type$, and $cluster_2$ refers variables $airconditioning$ and $dustfilter$. c_1 contains two variables, $model$ and $fuel\_type$, so c_1 is associated to $cluster_1$. Similarly, c_3 is associated to $cluster_2$. We then use Boolean variables to encode literals: a for $model = m_1$, b for $model = m_2$, c for $fuel\_type = petrol$, d for $fuel\_type = diesel$, and e for $fuel\_type = LPG$. We use f, g, h to encode, resp., the choices $manual$, $auto$, $none$ of $airconditioning$, and use i, j to encode, resp., the choices $with$, $none$ for $dustfilter$. We calculate the partial solutions for each cluster and get the $cluster_1$ and $cluster_2$ as in Figure 1.
- **Construction of the cluster tree**: A cluster tree is a tree whose nodes are clusters. An arc between two clusters shows a dependency between the linked clusters as a set of constraints. Hence, these latter involve variables in both clusters. It is encoded as a data structure that evaluates whether the partial solutions in the linked clusters are consistent w.r.t. these constraints. Thus, the structure allows to restore the complete configurations from the partial solutions in each cluster. Figure 2 presents the cluster tree for the example.

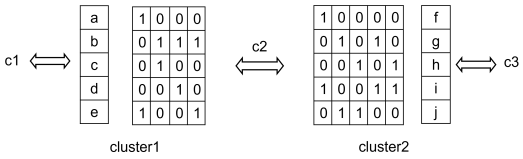

Fig. 2: Cluster tree example.

Indeed, for the same problem, many cluster trees can be derived, depending on the way of dividing variables to clusters. In order to optimize the cluster tree size, the current system builds the cluster tree with an heuristic analyses over the constraint graph [15], [16] and defines a dispatch of the different variables into the clusters of the tree.

- **Propagation on the cluster tree:** Whenever the state of a cluster changes (for example, a variable is assigned by the application request), it will make a global propagation through the whole cluster tree, and keep a deductively complete truth maintenance system.

The offline part saves the cluster tree in the memory for the use of the online part. Figure 3 give a global description of the current configuration system. The *Configuration Requests Management* activity takes the configuration request, looks for a matching solution in the "stored" configuration space, and returns the (positive or negative) answer.

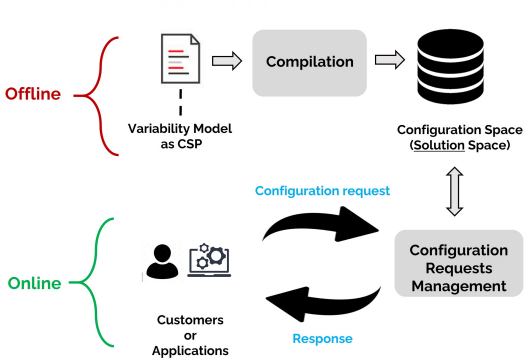

Fig. 3: Renault's Current Configuration System.

b) Challenge and Objective: this knowledge compilation based method invests once in a heavy offline level, and benefits a high-performing configurator when considering online requests. However, the growing variability and complexity of the range of models worsen the pressure on the memory storage.

Actually, operational applications need to keep hundreds of different compiled models in the RAM cache to guarantee a rapid response time for requests ($< 10ms$). Since the size of one compiled model can reach up to 800 Megabytes, the cumulative size of the compiled models in the cache could be tens of Gigabytes or more. As the actual servers don't always have such memory resources, reducing the memory size of individual compiled models, while not degrading response time is an important challenge to cope with.

c) Motivation of exploring symmetries: observing the cluster structure, we can notice that columns in the matrix will become unnecessary if they can be deducted from others. With this idea in mind, symmetries, which represent the isomorphic parts in the configuration space, seem to be an interesting direction to explore.

d) Constraints of our contribution: To avoid any possible confusion, we specify here, once more, our working constraints. Since the current configuration system is used by plenty of operational applications, a radical change in the used technology can bring potential risks. Hence, in our work, the system is taken as a black (or even grey) box and improvements are made as pre- and post-treatments. In details, the working constraints are the following: (i) A fixed input format: the CSP instances we work on are encoded in the XCSP3 format [17], generated by Renault from real industrial variability models. (ii) A fixed configuration space compilation process: the current compilation process and the choice of using cluster tree as the data structure shouldn't be changed.

III. Leveraging Symmetries to optimize the configuration system of Renault

Symmetries are very common in car industry. The interchangeability of *colors* mentioned in introduction can also exist for radios, roof types, etc. Here, we aim at exploiting this property to help reduce the configuration space representation.

A first key point is to be able to detect the presence of these isomorphic parts but at the level of the problem description. In practice, one needs to know if the set of solutions of the CSP at hand has isomorphic parts, without actually constructing the solutions. This can be achieved by studying the presence of *symmetries* in the structure of the CSP under study.

This section presents symmetries in the context of CSPs and their exploitation to reduce the configuration space. It then illustrates the effects of symmetries with the example of Section II-B. We conclude the section by giving the numbers on the symmetries we detected in the datasets.

A. Symmetries in CSP

A *solution symmetry* of a CSP is a permutation of its *literals* that preserves the set of solutions [18]. It is a bijection from literals to literals that maps solutions to solutions [19]. Solution symmetries are classified into following categories:

- `Value symmetries` [19]: a value symmetry g is a symmetry of solution such that there exists, for a variable x, a permutation θ of $Dom(x)$, with $g.(x = d) = (x = \theta.d)$, and $d \in Dom(x)$. For example, consider a variable x with $Dom(x) = \{d_1, d_2\}$ and $\theta.d_1 = d_2$. If g is a value symmetry of the problem in question, i.e. $g.(x = d_1) = (x = \theta.d_1 = d_2)$, then for **all solutions** with $x = d_1$, the assignments obtained by exchanging, in these solutions, $x = d_1$ with $x = d_2$ and keeping all other variables with their values, are also solutions of the problem.
- `Variable symmetries` [19]: a variable symmetry is a solution symmetry g such that it exists a permutation σ of the variables such that $g.(x = d) = (\sigma.x = d)$, with $d \in Dom(x)$. Consider, for example, the assignment $\mathcal{A} = \{x_1 = 1, x_2 = 2, x_3 = 3\}$, that is a solution to the problem at hand, and a variable symmetry g: that is a permutation σ, with $\sigma.x_1 = x_2$, $\sigma.x_2 = x_1$ and $\sigma.x_3 = x_3$. Hence, and by extension, applying σ on $\mathcal{A}$ derives the assignment $\{x_1 = 2, x_2 = 1, x_3 = 3\}$, that is also of valid solution.

This paper will focus on value symmetries since these are the symmetries our approach currently leverages.

B. Detecting and Breaking symmetries in a CSP

Here, we give a general overview on how to detect and break symmetries in a CSP. The detailed usage of these latter (in our context) is discussed in section IV.

Computing the symmetries of a CSP reduces to solve a graph automorphism problem [20]. Hence, we need to encode the CSP as a colored graph and then apply a graph automorphism tool (e.g., saucy3 [21] , bliss [22]) to get the automorphisms. Symmetries are then a straightforward interpretation of the latter.

Once detected, one can exploit these symmetries to avoid exploring isomorphic parts in the search structure. Such an approach helps reduce memory usage as well as execution time. Exploiting symmetries can happen in different ways that fall into two categories: 1) Static symmetry breaking [20], and 2) Dynamic symmetry breaking [23], [24]. We give an overview of the first approach since our method relies on it.

Static symmetry breaking: Once the group of solution symmetries (of a CSP) is known, say G, then the solutions of the problem can be partitioned into *equivalence classes (EC)*. All solutions belonging to an EC can be obtained from each other by some element of G. Hence, to get all solutions of a class, one needs to know only one solution of the EC and then derive the others by applying the appropriate $g \in G$.

The central concept is to augment the CSP with a set of constraints, called *Symmetry Breaking Constraints (SBCs)*, which preserves *very few solutions* per class and discards all others[1]. Thus, the problem's solution space considerably decreases while preserving all the necessary information to reconstruct the whole solution set.

To compute *SBCs*, the most classical way is to apply the *lex-leader method* [20]. It requires a static ordering on literals (we use the symbol $\prec$ to denote this ordering). For example: $(x_1 = 1) \prec (x_1 = 2) \prec (x_2 = 1) \prec$ ... With such an

[1]Ideally, one wants to preserve only one solution per class, but this is computationally very expensive!

ordering, one can establish a total lexicographical order on the solutions of the problem at hand. We then define the min (or max) solution, called *canonical* solution, for each EC as the one we preserve. The generated *SBCs* are constructed such that only canonical solutions can satisfy them.

For example, consider the detection of a solution symmetry g such that: $g.(x = d) = (x = \theta.d = d')$. By imposing the ordering $(x = d) \prec (x = d')$, we augment the problem with the following *SBC*: $\neg(x = d')$. This constraint will be satisfied by all the solutions where $x = d$ and not satisfied by those where $x = d'$. Here, we preserve *equi-satisfiability* of the original and the augmented problems while reducing the memory footprint and/or the solving time of the latter.

It is worth noting that the equi-satisfiability does not imply logical equivalence. As discussed before, the additional *SBCs* in the augmented problem will discard all the solutions of the same EC but a few, which has a side effect on our approach. We will discuss that issue in the remainder of the paper.

C. Value symmetries in action

Back to the example of Section II-B to explain the effects of value symmetries. The solution space of p is:

- s_1= $\{m = m_1, ft = LPG, ac = manual, df = with\}$
- s_2= $\{m = m_1, ft = LPG, ac = auto, df = none\}$
- s_3= $\{m = m_1, ft = LPG, ac = none, df = with\}$
- s_4= $\{m = m_1, ft = LPG, ac = auto, df = with\}$
- s_5= $\{m = m_1, ft = LPG, ac = none, df = with\}$
- s_6= $\{m = m_2, ft = petrol, ac = auto, df = with\}$
- s_7= $\{m = m_2, ft = petrol, ac = auto, df = none\}$
- s_8= $\{m = m_2, ft = diesel, ac = auto, df = with\}$
- s_9= $\{m = m_2, ft = diesel, ac = auto, df = none\}$
- s_{10}= $\{m = m_2, ft = LPG, ac = auto, df = with\}$
- s_{11}= $\{m = m_2, ft = LPG, ac = auto, df = none\}$
- s_{12}= $\{m = m_2, ft = LPG, ac = none, df = with\}$
- s_{13}= $\{m = m_2, ft = LPG, ac = none, df = none\}$
- s_{14}= $\{m = m_2, ft = LPG, ac = manual, df = with\}$

(m represents $model$, ft represents $fuel\_type$, ac represents $airconditionning$, df represent $dustfilter$)

a) Value symmetries identification: we can observe that for each solution where $ft = petrol$ (s_6, s_7), we have a symmetrical solution where $ft = diesel$, with all the other variable values being the same (s_8, s_9). Hence, this is a value symmetry in p between the two values of the variable $fuel\_type$. Using a cyclic notation, this is written as: $((fuel\_type = petrol) \quad (fuel\_type = diesel))$

Indeed, to detect symmetries in a CSP, it actually needs calculating the graph automorphism of the colored graph encoding the CSP, which is discussed in section IV-A.

b) Breaking value symmetries: in the example above, we can get s_8 from s_6, and get s_9 from s_7 by permuting the values of $fuel\_type$. Hence, we don't need to save all the solutions. We add to the problem a set of *SBCs* to remove the symmetric solutions. To do so, we fix the ordering on *literals*. Since only $fuel\_type$ is involved in the value symmetries, we need an ordering on the literals, as the following: $(fuel\_type = petrol) \prec (fuel\_type = diesel) \prec (fuel\_type = LPG)$. So, the following *SBC* enriches the original problem: $c_4 \equiv \neg(fuel\_type = diesel)$.

Therefore, the solutions of the new problem reduce to a set of 12 solutions with removing solution s_8 and s_9. We build the cluster tree for the new problem with the *SBC* c_4 added. According to c_4, Boolean variable d should always be $false$. So the new cluster tree will be as shown in Figure 4: $cluster_1$ in Figure 2 is compressed by c_4 and then we get $cluster'_1$ instead.

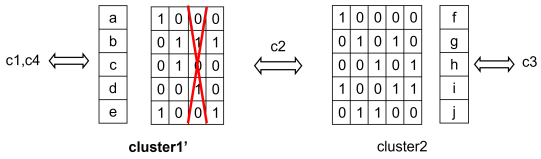

Fig. 4: New cluster tree.

Now the solutions space has been successfully reduced and the the cluster tree is well compressed too. This simple example helps give a general idea of how symmetries can work and be used to reduce size of the cluster tree.

D. Value symmetries in Renault's CSP instances

To complete our discussion about symmetries, we present, in Table I, the results on the symmetry profile exhibited by the datasets of Renault. In Table I, $\#CSP$ is the total number of CSP instances in the datasets. $\mathrm{Avg}(\#P)$, $\mathrm{Min}(\#P)$, $\mathrm{Max}(\#P)$, $\mathrm{SD}(\#P)$ represent, respectively, the average, minimum, maximum and standard deviation value of permutation numbers detected in the datasets.

TABLE I: Number of value symmetries detected in the datasets of Renault .

$\#CSP$	$\mathrm{Avg}(\#P)$	$\mathrm{Min}(\#P)$	$\mathrm{Max}(\#P)$	$\mathrm{SD}(\#P)$
424	317	14	2217	224

We observe that the number of value symmetries detected is quite large and this justifies the interest of the proposed approach. The following section describes a full implementation of our approach around the current configuration system. In addition, we explain how to detect value symmetries and give formal algorithms for generating *SBCs* and canonical queries.

IV. Revisiting the Product Configuration System

This section details how we operate each activity in the new configuration system. The following two figures use a different color to specify the activities we add.

Figure 5 highlights the revisited Offline level of the configuration system with the integration of two new activities described earlier: 1) *Value Symmetries Detection*, which provides the *Symmetries Information* inside the configuration space. 2) *Value Symmetries Breaking* that augments the CSP with *SBCs*.

Figure 6 shows how the revisited Online level operates, by considering the *Symmetries Information* during the *Configuration Requests Management*. We propose two new activities:

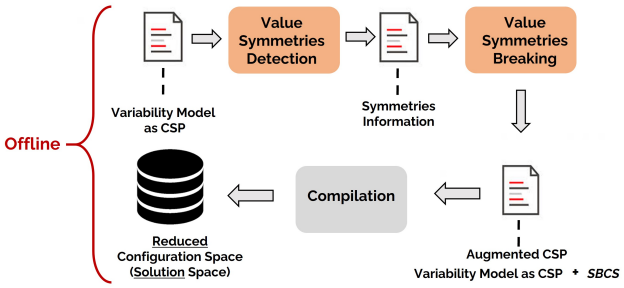

Fig. 5: The new configuration system - The Offline level.

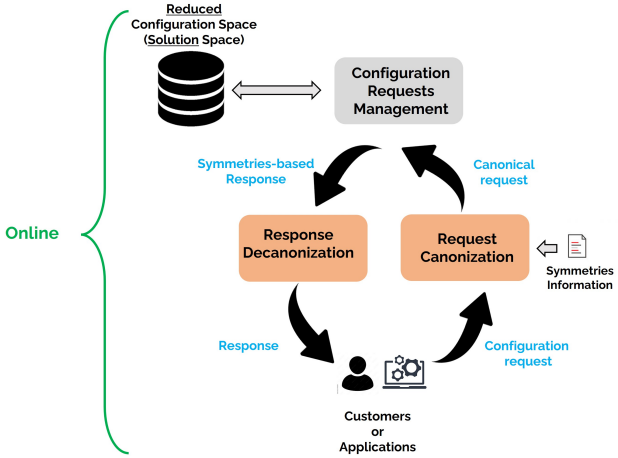

Fig. 6: The new configuration system - The Online level.

- ***Request Canonization:*** since the reduced solution space contains only canonical solutions, then to correctly answer a *configuration request*, the request must be canonized. This activity computes the equivalent (symmetrical) representation of the given request using the lex-leader method. We refer to this activity as *Request Canonization*, which takes the original request and the *symmetries information* to calculate the *canonical request*. Then the system takes this new *canonical request* and searches in the *Reduced Configuration Space* to find the solutions.
- ***Response Decanonization:*** when responding to the customer or applications, if we only need to respond whether the *configuration request* is satisfiable or not, then the responses for the original *configuration request* and its *canonical request* are the same. We can directly pass the *symmetries-based Response* as the final response to the customer. However, if the customer also asks for the satisfying solutions, decanonizing the response is necessary because the satisfying solutions in the *symmetries-based Response* respond to the *canonical request*, but not to the original *Configuration request*. Inside the solutions, the literals involved in symmetries are replaced by permutable literals that have a lower order. Therefore, we need the activity *Response Decanonization* to decanonize the found solutions, and return the decanonized response.

The following sections detail the four new activities above.

A. Value Symmetries Detection Activity

To compute the symmetries of a CSP, we need to encode the CSP as a colored graph and then calculate the graph automorphism [20].

a) Encoding the colored graph: the choice of the used colored graph is very important. An improper graph choice can lead to the loss of symmetries information or bring unexpected difficulties. For example, one colored graph proposed in [19] doesn't contain value symmetries since this graph encodes equally *extension* or *intension* constraints as nodes, and does not encode enough details on the literals. Another graph called *variable-value graph* in [19] turn out to be extremely large for our instances. This graph needs to encode each allowed assignment of each constraint as a node. To calculate the allowed assignments of *intension* constraints, one needs to rewrite them under the form of *extensions*. Our experiments showed that, in our datasets, an *intension* constraint can have tens of thousands of allowed assignments. Therefore, this leads to a graph with one constraint node linked to tens of thousands of assignment nodes, which is too large to deal with.

Our graph encoding approach relies on the one proposed by Mears [25], itself based on the Puget's *variable-value graph* [19]. The particularity of Mears' graph is that, while rewriting the *intension* constraints into *extension* constraints, one can choose to compute the **allowed** or **not allowed** assignments. The resulting graph still allows the detection of value symmetries while being less memory-intensive: complicated *intension* constraints, with more than ten thousand allowed assignments, have only hundreds of not allowed assignments.[2]

b) Computing the graph automorphisms: Once the studied CSP is encoded, we calculate the automorphisms of the resulting graph using *bliss* tool [22]. The output is a set of permutations between the node identifiers. We straightforwardly get the value symmetries we are looking for by decoding node identifiers into literals.

It's worth noting that value symmetries extracted by *bliss* can be arbitrary complex (binary permutations, ternary permutations, etc). So, exploiting these symmetries can quickly become complicated to handle. Since our experiments show that 95% of the detected symmetries are binaries[3], we decide to focus only on those symmetries and develop our algorithms according to this restriction.

B. Value Symmetries Breaking Activity

With the value symmetries obtained above, we calculate the *SBCs*. Algorithm 1 describes the procedure of *SBCs* calculation using the *lex-leader* method. The generated *SBCs* are added to the original CSP to get the augmented CSP.

Algorithm 1 goes through the permutations given as input (line 3), compares the order of the literals for each permutation and generates the corresponding SBC (lines 5 and 7).

[2]The rewriting of the *intension* constraints into *extensions* is done using the solvers `abscon` [7].

[3]Under cyclic notation, these are permutations of the form $((x = v_1)(x = v_2))$.

Algorithm 1 SBCs-calculation(*P,O*): returns a set of *SBC*, taking *P*, the set of permutations, and *O*, the global literal ordering as inputs.

```
1: procedure SBCs-CALCULATION(P, O)
2:     SBCs=[]
3:     for ((x = d)(x = d')) ∈ P do
4:         if (x = d) ≺ (x = d') then
5:             SBCs.add(not(x = d'))
6:         else
7:             SBCs.add(not(x = d))
8:         end if
9:     end for
10:    return SBCs
11: end procedure
```

C. Request Canonization Activity

The general idea of canonizing requests is as follows: we iterate on each *literal* of an incoming request, verifying whether it is involved in a value symmetry; if not, we keep it; otherwise, we replace it with the symmetrical literal having the least lexicographical order. Algorithm 2 formalizes this process. We first sort the literals in r_o according to the global order *O* (line 3), then for each literal in r_o, the procedure goes through all the permutations (line 4). If a literal is smaller than the literal at hand, a swap takes place (lines 6-10). The resulting request is the canonical one (line 14).

Algorithm 2 Requests canonization(r_o, *P*, *O*): returns r_c, the canonical form of the input request (r_o), taking the original request r_o, *P* the set of permutations, and *O* the global literal ordering as inputs.

```
1: procedure REQUESTS-CANONIZATION(r_o, P, O)
2:     r_c = []
3:     r_o = sort(O, r_o)
4:     for l_o ∈ r_o do
5:         l_r = l_o
6:         for (l, l') ∈ P do
7:             if (l_r == l && l' ≺ l) then
8:                 l_r = l'
9:             else if (l_r == l' && l ≺ l')
10:                l_r = l
11:            end if
12:        end for
13:        r_c.add(l_r)
14:    end for
15:    return r_c
16: end procedure
```

D. Response Decanonization Activity

The response decanonization process is simple: if canonized request r_c is unsatisfiable, we respond directly with "unsatisfiable". Otherwise, we replace the values of the variables involved in symmetries with the values in the original request.

V. Experiments and results

This section presents the experimental results with the revisited product configuration system. First, we present the datasets. Then we present the experiments we conduct: (i) symmetry detection; (ii) offline process: compilation of the original and augmented CSPs; (iii) and online process: response to configuration requests using the original and the new systems. All experiments are carried out on a machine with an Intel Xeon CPU 2.80GHz with 16GB of memory.

A. Datasets

We collect 424 CSP instances each representing a variability model. These instances are used at an intermediate stage of the commercial offer, representative of the real range of vehicles at a given time, and covering a good part of the diversity cases of Renault. These CSP instances are encoded in the XCSP3 format [17]. Variables' domains are finite sets of integers, and each CSP contains both *extension* and *intension* constraints.

To better evaluate the contributions of our approach, we classified the instances into several groups. We used for that a custom indicator provided by Renault. This indicator is linked to the offline compilation process. It's used to evaluate the instance solving difficulty within the compilation system. The calculation of this indicator is based on the number of cycles detected in the constraint graph [3]. We calculate the number of connected component (note as Ncc) for each graph. A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path [26]. Then for each connected component, we use tarjan's cycle enumeration algorithm [27] to calculate the number of cycles (Note as Nci). We also compute the number of nodes for each connected component (Note as $Nnode$). Finally we use the sum of the product of Nci and $Nnode$, (noted as $\sigma = \sum_{i=1}^{Ncc} (Nci_i \times Nnode_i)$), as indicator to classify the models.

We group the instances according to σ. Table II shows the four obtained classes. Each class contains 106 CSPs. Column $Avg(\sigma)$ represents the average σ value for each class. $Min(\sigma)$ (resp. $Max(\sigma)$) represents the minimum value (resp. maximum value) of σ for each class.

TABLE II: Subdivision of the datasets in four classes of CSPs using the calculated σ.

Class	#CSP	Avg(σ)	Min(σ)	Max(σ)
C_1	106	8,141	2	33,969
C_2	106	167,331	34,957	429,476
C_3	106	1,860,927	442,936	4,009,127
C_4	106	239,425,467	4,098,726	12,447,733,956

B. Measuring time of generating augmented CSPs with SBCs

The first experiment concerns measuring the time of activities *Value Symmetries Detection* and *Symmetries Breaking*.

Table III presents the results of *Value Symmetries Detection* with the number of value symmetries detected for each class. We observe that the average numbers of value symmetries detected for each class increases along with the solving difficulty. Table IV presents the execution time (unit: minutes)

TABLE III: Number of value symmetries detected for each class in the datasets.

Class	Avg(#P)	Min(#P)	Max(#P)	SD(#P)
C1	151	14	492	80
C2	320	46	900	139
C3	363	58	647	106
C4	415	22	2,217	315
Total	317	14	2,217	224

TABLE IV: Execution times (unit: minute) for generating the augmented CSPs with SBCs for each class in the datasets.

Class	Sym. detection		Sym. breaking	Total
	In2Ex	bliss		
C_1	114	0.40	0.02	114.42
C_2	323	3.47	0.05	326.52
C_3	292	9.25	0.04	301.29
C_4	670	13.78	0.07	683.85
Total	1,399	26.9	0.18	1,426.08

of both activities. Column **In2Ex** represents the total execution time for the conversion of *intension* constraints to *extensions* during *Value Symmetries Detection*. Column **bliss** represents the total time for calculating graph automorphisms. Column **Sym. breaking** represents the total time for calculating *SBCs*.

We observe that the step of transforming *intension* constraints to *extensions* consumes the most time. All the other steps execute quickly, especially the calculation of *SBCs* (column **Sym. breaking**).

We also observe that the total execution time (1426 mins) is relatively high. However, all these steps happen offline and are separated from the compilation activity (see Figure 5). Besides, the cost could be absorbed if the obtained reduction is significant: (i) allowing the compilation of CSPs that could not be treated before; (ii) allowing the reduction of the response time to customers' queries.

C. Measuring the compilation time and memory usage of the compiled structure

In this experiment, we use the current compilation system to compile both the original CSPs and the CSPs with *SBCs*. Hereafter, we report the resulting structures' compilation time and memory usage for both types. Tables V and VI presents the results, followed by our observations and conclusions.

TABLE V: Memory usages (unit: MB) of the built solution spaces by compiling the original (CM_o) and the augmented CSPs (CM_a).

Class	CM_o	CM_a	Δ_m
C_1	8.13	5.00	-38.5%
C_2	3,952	1,857	-53.01%
C_3	10,671	4,735	-55.62%
C_4	7,667	4,075	-46.8%
Total	22,298.13	10,672	-52.13%

Table V is the resulting memory usages: CM_o (resp. CM_a) represents the total memory usage (in MB) for each class of the original (resp. the augmented with SBCs) CSPs. Δ_m reports the difference between CM_a and CM_o in percentage: $\Delta_m = (CM_a - CM_o)/CM_o$.

TABLE VI: Compilation times (unit: minute) of the original (CT_o) and the augmented (CT_a) CSPs, excluding the time for symmetry detection. Δ_t expresses the gain or the loss.

Class	CT_o	CT_a	Δ_t
C_1	0.12	0.13	8.33%
C_2	7.73	4.91	-36.48%
C_3	33.76	17.60	-47.87%
C_4	27.24	19.24	-28.15%
Total	68.85	41.88	-39.18%

Table VI presents the compilation times (unit: minute)[4]: CT_o (resp. CT_a) represents the total compilation time for the original (resp. augmented) CSPs. Δ_t shows the difference between CT_a and CT_o in percentage: $\Delta_t = (CT_a - CT_o)/CT_o$.

From Tables V and VI, we observe that: (i) For all the classes, our new configuration system reduces the memory usage. The memory reduction is up to 55.62%, w.r.t. the original approach. Additionally, the new configuration system reduces the compilation time by up to 47.87%. (ii) For simple classes (C_1), the compilation time remains almost the same.

Our first conclusion is that the introduction of symmetries could be beneficial to treating very complex VMs and hopefully manage previously intractable problems.

We believe our method has a positive impact over all the classes of instances. Especially, for the complex classes, the SBCs performs quite well. The reason why it takes more time to compile the instances of C_1 is their relative simplicity. Actually, instances of C_1 do not contain that much of value symmetries, and the integration of SBCs does not simplify their solving. However, for the complex classes, the added SBCs decrease the configuration space dramatically, since we have to store only few represents for each EC.

D. Measuring configuration requests' response time

Configuration requests can be quite variant in real life. There are several basic types of requests [28]: (i) a satisfiability check; (ii) compute the number of satisfying solutions for requests; (iii) compute all valid solutions for requests.

In our benchmark, the configuration request type we used is the first, a satisfiability check. These requests are the partial configurations consisting of assignments of values to a certain variables, in order to check whether requests are satisfiable. The general form of these requests is then a conjunction of variable assignments. The number of variables in each request varies from 2 to the total number of variables of the CSP at hand. For each instance, we generate 2000 requests: 1000 are satisfiable, and 1000 are not. This choice is motivated by the fact that it simulates the actual number and type of requests run by certain operational applications of Renault. Actually, this type of requests is used for the forecast product offer.

Indeed, the request form can be more than the conjunction of assignments. One can create its request with different operators (disjunction, implication, etc.), to describe the relations between the variable assignments. Since these requests can be transformed into a set of requests of the conjunction form, we only consider the conjunction form request in the test.

[4]Without considering the time for symmetry detection.

We report here the requests' response times for the original and reduced configuration spaces (for each class of problems). The former is the time of searching for solutions over the original configuration space. The latter includes the times of the activity *Request Canonization* and searching for solutions over the reduced configuration space. Since the request is a simply satisfiability check, the activity of *Response Decanonization* is not taken into consideration.

Table VII presents the results: **nb_req** is the total number of requests (number of instances times number of requests for each instance), t_o (unit: second) is the total response time of the original system, and t_a is the response time of the new system (including canonization and interrogation execution times). Δ_t reports the difference between t_a and t_o.

TABLE VII: Requests' response times (unit: second) for the current (t_o) and the new (t_a) configuration systems.

Class	nb_req	t_o	t_a	Δ_t
C_1	212,000	14.24	20.04	40.73%
C_2	212,000	513.83	243.07	-52.69%
C_3	212,000	1,530.8	714.4	-53.33%
C_4	212,000	1,420.5	1081.6	-23.86%
Total	848,000	3,479.37	2,059.11	-49.81%

We observe that, on complex problems (classes other than C_1), the gain achieved by the new system is significant (up to 53%, compared to the original system). For the class where the problems are relatively simple, the new system performs poorly and worsens the response time (compared with the old system). It turns out that the loss is due to the canonization step, as highlighted in Table VIII.

TABLE VIII: C_1 requests' response times detailed. t_o and t_a (unit: second) are the same as in Table VII, t_{ac} is the canonization time, and t_{ai} is the solution searching time over the reduced configuration space.

Class	t_o	t_a	t_{ac}	t_{ai}
C_1	14.24	20.04	5.7	14.334

We notice that t_{ai} is almost equivalent to t_o. This is reasonable because the size of the reduced configuration has slight difference compared to the original (as in Table V). t_{ac} is the reason of the loss. But we can also easily calculate that on average, it takes only around 0.03ms $((t_a - t_o)/212000)$ plus to deal with each request.

We can then conclude: 1) For simple classes, the cost of canonization worsens the response time w.r.t to the original approach. Nevertheless, this degradation is negligible because it results in a response time of less than 0.1 ms for each request. 2) For complex classes, the reduction is so dramatic that the cost of canonization is entirely absorbed by the short time needed to search in a small structure.

E. Summary

With all experiments together, we conclude the following:

- **Long pre-processing time:** Computing symmetries can be very long. However, since it is an offline phase, our requirements in terms of computation time can be less stringent. Nevertheless, this phase should be optimized.
- **Reduction of memory usage:** Our configuration system succeeds in reducing the memory usage for complex classes. The achieved gain is of 52% on average.
- **Reduction of compilation time:** While the preprocessing time is relatively high, the compilation of the augmented CSPs is small and fulfills the requirements.
- **Reduction of requests' response times:** The experiment in section V-D shows that our method can dramatically save the response time on the requests involving complex instances (50% on average compared with the original approach). However, the approach has a rather negative impact when considering simple instances, which results in nuances in the approach's usefulness in the general case. Indeed, it is unnecessary to add the complexity of the use of symmetries when the processed instances are simple enough, and their solution spaces are easily stored.

To reproduce our experiments, we provide a file available at: https://zenodo.org/record/7538421, including: (i) examples of VMs specified as CSPs. For confidentiality reasons, we only provide anonymized data; (ii) the source code of our tool for detecting symmetries on CSPs that can be easily reused in other contexts; (iii) examples of the value symmetries of the provided VMs.

VI. RELATED WORK

*a) **Mapping Variability Models to CSPs**:* the mapping from variability models to CSPs has been widely discussed in the literature [4], [28]–[31]. Among them, many works consider the specific case where the variability models are represented as feature models [4]. Actually, several extensions of the CSP paradigm handle the modeling of constraint-based range of products. For example, Dynamic CSP [32] is often used in the case where the existence of optional variables depends on the value of another variable. Other extensions include composite CSP [33] and interactive CSP [34].

In this study, since the characteristics taken into account by these formalisms are not crucial for Renault , a classical CSP is very well suited to represent the product range at hand.

*b) **Knowledge Compilation**:* as discussed earlier, the configuration system studied here relies on knowledge compilation. *Darwiche and P. Marquis* have provided a detailed comparison between different representation languages in [8] to compare their efficiency in supporting different types of queries (for example, polynomial-time consistency check). In addition, many other languages have also been proposed (Set-Labeled DD [35], SDD [10]) recently.

The representation language used by the configuration system of Renault differs from those studied by *Darwiche and P. Marquis*. We are working on formalizing this representation and integrating it in the global hierarchy presented in [8].

*c) **Symmetries**: Gent et al.* have discussed the symmetries in CSPs in [36]. Specifically, the related work in this domain mainly differs from the symmetry breaking method and the types of symmetries to explore. For the former, as we mentioned earlier, the dynamic symmetry breaking method

has been studied by many researchers (SBDS [23], STAB [24], SBDD [37]). Moreover, for the latter, Puget has proposed how to detect and benefit from variable symmetries in [19].

Back to our work, integrating symmetries dynamically will require a full access to the system. This will be difficult for us because of our working constraint. But exploring the other types of symmetries seems to be a next promising direction.

VII. Conclusion

This paper presents an approach that exploits symmetries in configuring model variability for Renault. The aim is to tackle the growing complexity of the vehicles' range and improve the performance of the configuration system. We have shown how symmetries can be loosely coupled to this system while achieving dramatic savings in memory and time usage.

Our first perspective is to improve the process of detecting symmetries. The main problem is the used time to convert *intension constraints* to *extensions*. Parallelism is a straightforward way for this problem. Indeed, the constraints can be converted in parallel with minimal effort. Furthermore, as we detected other types of symmetries, namely, variable symmetries and variable-value symmetries [25], in our datasets, we plan to extend our work to handle these symmetries. The issue is to derive a new algorithm for generating *SBCs* since the one in this paper is not straightforwardly enforceable for such symmetries. It will be also interesting to evaluate the performance of our approach over the other open source data sets. Another point is investigating how symmetries can be integrated directly into the compilation process, which can help guide the internal algorithms to derive more compact structures and improve the performance even further.

References

[1] T. B. et al., "A survey of variability modeling in industrial practice," in The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013, S. Gnesi, P. Collet, and K. Schmid, Eds. ACM, 2013, pp. 7:1–7:8.

[2] M. Cordy and P. Heymans, "Engineering configurators for the retail industry: experience report and challenges ahead," in Proceedings of the 33rd Annual ACM Symposium on Applied Computing, 2018.

[3] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," European journal of operational research, vol. 119, no. 3, pp. 557–581, 1999.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," Information Systems, vol. 35, no. 6, pp. 615–636, 2010.

[5] S. A. Cook, "The complexity of theorem-proving procedures," in Proceedings of the Third Annual ACM Symposium on Theory of Computing, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158.

[6] C. Prud'homme, J.-G. Fages, and X. Lorca, "Choco solver documentation," TASC, INRIA Rennes, LINA CNRS UMR, pp. 13–42, 2016.

[7] C. Lecoutre and S. Tabary, "Abscon 109: a generic csp solver," 01 2008.

[8] A. Darwiche and P. Marquis, "A knowledge compilation map," Journal of Artificial Intelligence Research, vol. 17, pp. 229–264, 2002.

[9] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in 27th ACM/IEEE design automation conference. IEEE, 1990, pp. 40–45.

[10] A. Choi and A. Darwiche, "Dynamic minimization of sentential decision diagrams," in Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA, M. desJardins and M. L. Littman, Eds. AAAI Press, 2013.

[11] K. A. Sakallah, "Symmetry and satisfiability." Handbook of Satisfiability, vol. 185, pp. 289–338, 2009.

[12] J. Astesana, L. Cosserat, and H. Fargier, "Constraint-based vehicle configuration: A case study," in ICTAI (1). IEEE Computer Society, 2010, pp. 68–75.

[13] B. Pargamin, "Vehicle sales configuration: the cluster tree approach," in ECAI 2002 Configuration Workshop, 2002, pp. 35–40.

[14] P. Bernard, "Extending cluster tree compilation with non-boolean variables in product configuration: A tractable approach to preference-based configuration," in Proceedings of the IJCAI, vol. 3. Citeseer, 2003.

[15] E. Amir, "Efficient approximation for triangulation of minimum treewidth," arXiv preprint arXiv:1301.2253, 2013.

[16] A. Becker and D. Geiger, "A sufficiently fast algorithm for finding close to optimal clique trees," Artificial Intelligence, no. 1-2, pp. 3–17, 2001.

[17] O. Roussel and C. Lecoutre, "Xml representation of constraint networks: Format xcsp 2.1," arXiv preprint arXiv:0902.2362, 2009.

[18] D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith, "Symmetry definitions for constraint satisfaction problems," Constraints, vol. 11, no. 2-3, pp. 115–137, 2006.

[19] J.-F. Puget, "Automatic detection of variable and value symmetries," in International Conference on Principles and Practice of Constraint Programming. Springer, 2005, pp. 475–489.

[20] F. Rossi, P. Van Beek, and T. Walsh, Handbook of constraint programming. Elsevier, 2006.

[21] H. Katebi, K. A. Sakallah, and I. L. Markov, "Symmetry and satisfiability: An update," in International Conference on Theory and Applications of Satisfiability Testing. Springer, 2010, pp. 113–127.

[22] T. Junttila and P. Kaski, "Engineering an efficient canonical labeling tool for large and sparse graphs," in 2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, 2007, pp. 135–149.

[23] I. P. Gent, W. Harvey, and T. Kelsey, "Groups and constraints: Symmetry breaking during search," in International Conference on Principles and Practice of Constraint Programming. Springer, 2002, pp. 415–430.

[24] J.-F. Puget, "Symmetry breaking using stabilizers," in International Conference on Principles and Practice of Constraint Programming. Springer, 2003, pp. 585–599.

[25] C. Mears, M. G. De La Banda, and M. Wallace, "On implementing symmetry detection," Constraints, vol. 14, no. 4, pp. 443–477, 2009.

[26] J. Clark and D. A. Holton, A first look at graph theory. World Scientific, 1991.

[27] R. Tarjan, "Enumeration of the elementary circuits of a directed graph," SIAM Journal on Computing, vol. 2, no. 3, pp. 211–216, 1973.

[28] R. Pohl, K. Lauenroth, and K. Pohl, "A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models," in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011.

[29] F. A. et al., "Intelligent support for interactive configuration of mass-customized products," in International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems. Springer, 2001, pp. 746–756.

[30] H. A. et al., "Unifying software and product configuration: A research roadmap," 2012.

[31] D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank, "Automated analysis in feature modelling and product configuration," in International conference on software reuse. Springer, 2013, pp. 160–175.

[32] S. Mittal and B. Falkenhainer, "Dynamic constraint satisfaction," in Proceedings eighth national conference on artificial intelligence, 1990, pp. 25–32.

[33] D. Sabin and E. C. Freuder, "Configuration as composite constraint satisfaction," in Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop. AAAI Press Palo Alto, CA, 1996.

[34] E. Gelle and R. Weigel, "Interactive configuration using constraint satisfaction techniques," in Proceedings of PACT-96, 1996, pp. 37–44.

[35] A. Niveau, H. Fargier, and C. Pralet, "Representing csps with set-labeled diagrams: A compilation map," in Graph Structures for Knowledge Representation and Reasoning - Second International Workshop, GKR 2011, Barcelona, Spain, July 16, 2011. Revised Selected Papers, ser. Lecture Notes in Computer Science, M. Croitoru, S. Rudolph, N. Wilson, J. Howse, and O. Corby, Eds., vol. 7205. Springer, 2011, pp. 137–171.

[36] I. P. Gent, K. E. Petrie, and J.-F. Puget, "Symmetry in constraint programming," Foundations of Artificial Intelligence, pp. 329–376, 2006.

[37] T. Fahle, S. Schamberger, and M. Sellmann, "Symmetry breaking," in International Conference on Principles and Practice of Constraint Programming. Springer, 2001, pp. 93–107.