Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos

SOPORTE PARA LA CONFIGURACIÓN AUTOMÁTICA DE LÍNEAS DE PRODUCTOS

D. Héctor Pérez Morago

Dirigida por Dr. D. Ruben Heradio Gil
Dr. D. David Fernández Amorós

Departamento de Ingeniería del Software y Sistemas Informáticos
Escuela Técnica Superior de Informática
UNED

Ingeniería de Software cod.-31105128

Curso: 2013-2014
Convocatoria: Junio

Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos

SOPORTE PARA LA CONFIGURACIÓN AUTOMÁTICA DE LÍNEAS DE PRODUCTOS

D. Héctor Pérez Morago

Dirigida por Dr. D. Ruben Heradio Gil
Dr. D. David Fernández Amorós

Trabajo tipo A

Departamento de Ingeniería del Software y Sistemas Informáticos
Escuela Técnica Superior de Informática
UNED

Ingeniería de Software cod.-31105128 Madrid, Junio de 2014

Autorización

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16

28040, Madrid

Tel: 91 398 89 10

Fax: 91 398 89 09

www.issi.uned.es

Abstract

To compete in the global marketplace, manufacturers try to differentiate their products by focusing on individual customer needs. Fulfilling this goal requires companies to shift from mass production to mass customization. Under this approach, customized products are not designed individually but as a family of related derivatives. That is, a generic architecture, named product platform, is designed to support the derivation of customized products through a configuration process that determines which components the product comprises. When a customer configures a derivative, typically not every combination of available components is valid. To guarantee that all dependencies and incompatibilities among the derivative constituent components are satisfied, automated configurators are used. Flexible product platforms provide a big number of interrelated components, and so the configuration of all but trivial derivatives involves considerable effort to select which components the derivative should include. Our approach alleviates that effort by speeding up the derivative configuration using a heuristic based on the Information Theory concept of entropy. The effectiveness of the approach is empirically validated using a real case study taken from the automotive industry.

Keywords: Entropy Based Heuristic; Mass Customization; Product Configuration

Resumen

Para competir en un mercado globalizado, las empresas han de hacer un esfuerzo para diferenciar sus productos centrandose en las necesidades específicas de los clientes. Para alcanzar este objetivo, las empresas deben cambiar sus modelos de producción pasando de la *producción en masa* (mass production) a la *producción personalizada* (mass customization). Bajo esta nueva perspectiva, los productos personalizados no son creados individualmente sino como familias de productos relacionados. Aquí, una arquitectura genérica, llamada plataforma de productos (product platform), es diseñada con el fin de dar soporte a la derivación de productos personalizados mediante el proceso de configuración que indica que componentes tiene un producto. Cuando un cliente configura un producto (derivative) no todos las combinaciones de los componentes estan permitidas. Los configuradores son herramientas automáticas desarrolladas con el fin de garantizar que todas las dependencias e incompatibilidades entre componentes son satisfechas. Las plataformas de productos proporcionan un gran número de componentes interrelacionados, por lo que la configuración, de incluso los productos más triviales, conlleva un considerable esfuerzo. Nuestra propuesta alivia tal esfuerzo acelerando el proceso de configuración de un producto, utilizando para ello una heurística basada en el concepto de entropía de la teoría de información. La efectividad de nuestra propuesta ha sido empíricamente validada utilizando un caso de estudio real tomado de la industria automovilística.

Keywords: Mass Customization; Product Configuration; Entropy Based Heuristic

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

To increase variety, improve customer satisfaction, reduce lead-times, and shorten costs, many companies have shifted from *mass production* to *mass customization* [SSJ05]. This shift of paradigm enriches the mass production economies of scale with custom manufacturing flexibility by developing families of related products instead of single products. From this perspective, designing a product family is the process of capturing and modeling multiple product variants to satisfy different market niches. A generic architecture, named *product platform*, is designed to support the creation of customized products called *derivatives*.

Product platforms usually support a high quantity of derivatives. For instance, the number of derivatives for product platforms in the automotive industry may range from 10^3 for the smallest Peugeot and Nissan car models, to 10^{16} or 10^{21} for the BMW 3-Series and Mercedes C-Class, respectively [PH04]. To achieve that flexibility, a number of configuration options are available. For example, the Peugeot 206 and Mercedes C-Class car models have 86 and 389 customer selectable options, respectively. Typically not all option combinations are valid. There may be option incompatibilities (e.g.,"manual transmissions are not compatible with V8 engines"), option dependencies (e.g., "sport cars require manual gearbox"), etc. Configuring a valid derivative implies ensuring that all constraints between its constituent components are satisfied. Checking by hand those constraints is unfeasible for all but the most trivial product platforms, so derivative configuration is usually assisted by automated *configurators*[1] [SW98].

[1]Some examples of commercial configurators are *Configit* (http://www.configit-software.com/), *SAP Product Configurator* (https://scn.sap.com/docs/DOC-25224), *Oracle Configurator* (http://docs.oracle.com/cd/B12190_11/current/acrobat/115czinstg.pdf), etc. In addition, many automotive companies have their own configurators. For instance, *Volvo* uses *KOLA*, *Scania* uses *SPECTRA*, *Mercedes* uses *SMARAGD*, etc.

1. INTRODUCTION

Our work enriches existing configurators by reducing the number of steps required to configure a valid derivative. It takes advantage of the fact that, due to the component composition constraints, some decisions may be automatically derived from other decisions previously made. So the order in which decisions are made has a strong influence on the number of decisions required to complete a derivative. For instance, given the constraint “sport cars require manual gearbox” a customer might configure a sport car using two decision orderings: one requiring two steps (i.e., step 1: select “manual gearbox”, and step 2: select “sport car”), or another one using just a single step (i.e., select “sport car”, so decision select “manual gearbox” is implicitly made).

As van Nimwegen et al. [vNBvOS06] note, customers sometimes prefer to first answer questions that are important to them, or easy to answer, before being led through the remaining questions [vNBvOS06]. In this sense, our approach respects customer preferences. Instead of imposing a fixed ordering, it suggests orderings dynamically, reacting to the customer decisions. In particular, the process to get a derivative is performed in successive steps. In each step, the customer gets a question ranking, selects one of the questions and answers it. In the next step, the question ranking is readjusted to account for the customer’s answer. The computation of the ranking is grounded on the Information Theory concept of *entropy*, which was introduced by Shannon [Sha48] and measures the average uncertainty of a random variable.

At the first configuration step, the uncertainty is total. With no information at all, the configurator cannot figure out which derivative the customer desires. As the process advances, configuration options are eliminated according to the customer decisions and so the information about the final configuration increases (i.e., the set of included/excluded components grows). Consequently, the entropy decreases. When the derivative is totally configured there is no uncertainty and the entropy is zero.

As we will see, our approach and the heuristics proposed in [CE11] [MDSD14] require computing the probabilities of all variables in a Boolean formula. The usual way to perform such task is calling repeatedly a logic engine, e.g., a SAT solver or a Binary Decision Diagram (BDD) library, one time for each variable [KZK10]. Unfortunately, this approach has an high computational cost and thus imposes long response times, hindering customer-configurator interactivity. To overcome such problem, this paper proposes an algorithm that computes efficiently variable probabilities using BDDs. Since more complex logics than the Propositional one, which include integer arithmetic, transitive closure, etc., can be reduced to Boolean functions [HR04] [Jac12], and thus encoded as BDDs, our algorithm is general enough to support most configuration model notations.

The validity of our approach has been tested on two benchmarks widely used by the configuration and software product line communities: the Renault Megane platform provided by the car manufacturing company Renault DVI[1] and the Electronic Shopping case study [Lau06]. Results show that our approach requires less configuration steps

[1]`http://www.renault.fr/`

than related work, and that our BDD algorithm gets short response times, supporting this way not only our approach but also other methods proposed in related work.

The remainder of this paper is structured as follows. Chapter 2 presents the running example we will use to motivate and illustrate our work. Chapter 3 summarizes related work to our approach. Chapter 4 introduces the concept of entropy and describes how to compute it from a configuration model. Later, our entropy-driven approach is described in detail. Chapter 5 reports the experimental validation of our approach. Finally, Chapter 6 outlines the conclusions of our work.

CHAPTER

2

Motivational Example

This section illustrates the problem our approach tackles using an example provided by [WDSB09], where derivatives are cars with different automated driving capabilities.

To model the configurable options of a product family, a number of different notations are available. For instance, Feature Diagrams (FD) [KCH+90], Decision Diagrams [dec93], the Configit language, the SAP Product Configurator language, the Oracle Configurator language, etc. Interestingly, most of those notations are semantically equivalent [CGR+12, SHTB07]. In fact, automated configurators instead of processing configuration models directly, usually translate them into a propositional logic representation, such as a logic formula in conjunctive normal form, a BDD, etc. That logic representation is then processed using off-the-self tools, such as SAT solvers, BDD engines, etc. (see Section 4.1.2 for an explanation on the configuration model to logic translation). The input to our approach is the logic representation of the configuration model, so it is independent of the original notation used to specify the model.

To show what a configuration model looks like, please refer to Figure 2.1 which models our running example as a FD[1] (a hierarchically arranged set of features with different relations among them). Figure 2.1 includes three kinds of hierarchical relations:

- *optional*, denoted by simple edges ending with an empty circle; e. g., cars may (or may not) include an Automated Driving Controller (ADC).
- *mandatory*, denoted by simple edges ending with a filled circle; e. g., if a car has an ADC, it must include some kind of Collision Avoidance Braking (CAB).

[1]this paper follows the generic semantics for FDs given by Schobbens et al. [SHTB07].

- *alternative*, denoted by edges connected by an arc; e. g., Standard Avoidance (SA) and Enhanced Avoidance (EA) are the mutually exclusive options for Collision Avoidance Braking (CAB).

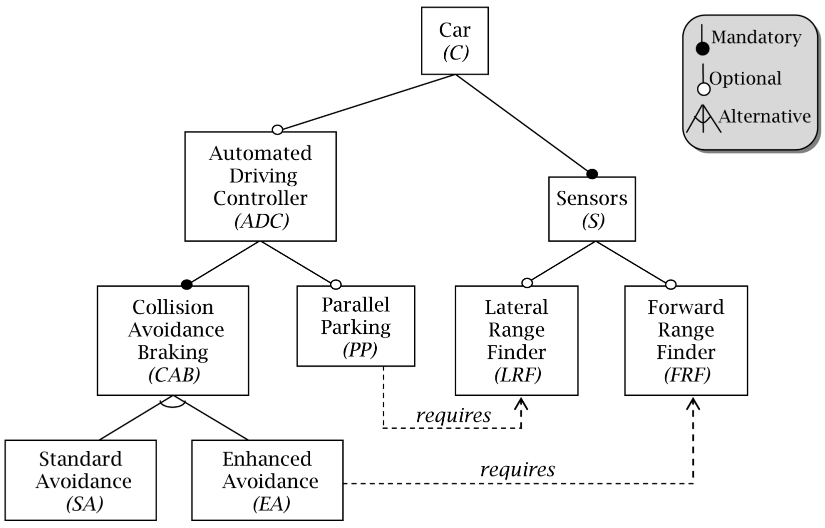


Figure 2.1: FD for car automated driving capabilities

To manage the complexity of modeling the similarities and differences among the derivatives of a product family, the FD notation follows a *divide and conquer* strategy. Derivative variabilities are modeled by progressively decomposing complicated and abstract features into simpler ones, until elemental features, which are directly implemented by physical components, are reached. The hierarchical structure of a FD graphically depicts such conceptual decomposition. From here on, derivatives will be expressed enumerating the final components they include, i.e., using references to the terminal nodes of the FD. For example, $\{$PP, LRF, FRF, $\neg$SA, EA$\}$ expresses the configuration of a car with components PP, LRF, FRF, EA and without SA.

The FD notation supports narrowing the configuration space by adding additional crosstree constraints. For instance, Figure 2.1 represents as "PP $\xrightarrow{\text{requires}}$ LRF" the fact that cars with Parallel Parking need to include the Lateral Range Finder component. Thus, a car derivative with components $\{$PP, $\neg$LRF, $\neg$FRF, SA, $\neg$EA$\}$ complies with the FD relations, but is not valid because violates the constraint "PP $\xrightarrow{\text{requires}}$ LRF".

For a configuration model with n options and no component interdependencies, the number of possible configurations is 2^n. Due to the feature relations and additional crosstree constraints, the number of valid configurations in the example is reduced from $2^5 = 32$ to the 13 ones summarized in Table 2.1.

To configure a car, the decision maker needs to answer a sequence of questions. For example, the sequence:

$$\boxed{\text{(1) } \textit{is EA in the configuration? no}, \text{ (2) } \textit{FRF? no}, \text{ (3) } \textit{LRF? yes}, \text{ (4) } \textit{PP? yes}}$$

Valid Derivatives	
1	¬PP, ¬LRF, ¬FRF, ¬SA, ¬EA
2	¬PP, ¬LRF, FRF, ¬SA, ¬EA
3	¬PP, LRF, ¬FRF, ¬SA, ¬EA
4	¬PP, LRF, FRF, ¬SA, ¬EA
5	¬PP, ¬LRF, ¬FRF, SA, ¬EA
6	¬PP, LRF, ¬FRF, SA, ¬EA
7	¬PP, ¬LRF, FRF, ¬SA, EA
8	¬PP, LRF, FRF, ¬SA, EA
9	¬PP, ¬LRF, FRF, SA, ¬EA
10	¬PP, LRF, FRF, SA, ¬EA
11	PP, LRF, ¬FRF, SA, ¬EA
12	PP, LRF, FRF, ¬SA, EA
13	PP, LRF, FRF, SA, ¬EA

Table 2.1: Valid derivatives for Figure 2.1

configures car 11 in Table 2.1. Current automated configurators guarantee the derivation of valid products ensuring the satisfaction of all model constraints. When the first question is answered, the configurator deduces that the car being configured necessarily includes SA (otherwise the alternative relation between EA and SA would not be hold). This way, the configurator is indirectly saving the decision maker from answering the irrelevant question *is SA in the configuration?*

The goal of our work is to make the most of the configuration model constraints going beyond current configurators to minimize the number of questions required to specify a derivative. To do so, our approach tries to find an optimal question ordering that maximizes the number of decisions automatically derived from other questions previously answered.

A straightforward approach to get such optimal question ordering is computing for each valid product all possible orderings, and thus finding the ordering with less questions on average for every product. Table 2.2 sums up the needed computations. For instance, the next-to-last column summarizes the number of questions needed for derivative {PP, LRF, FRF, SA, ¬EA}. Ordering $PP \prec LRF \prec FRF \prec SA \prec EA$ needs 3 questions, $LRF \prec PP \prec FRF \prec SA \prec EA$ needs 4, and so on. Afterwards, the average number of questions for each ordering is computed. Using this approach in the previous example, ordering $PP \prec LRF \prec FRF \prec SA \prec EA$ would be selected as an optimal one. As a result, the question sequence for derivative 11 in Table 2.1 would be shortened to:

(1) *is PP in the configuration? yes*, (2) *FRF? no*, (3) *SA? yes*

removing the need for answering *if LRF is in the configuration*.

Unfortunately, this approach requires $m \cdot n!$ computations, where n is the number of components of the configuration model and m is a number $\leq 2^n$. So it is extremely expensive in computational terms and does not scale except

orderings ($n!$)	**derivatives** ($\leq 2^n$)				**average number of questions**
	{¬PP, ¬LRF, ¬FRF, ¬SA, ¬EA}	{¬PP, ¬LRF, FRF, ¬SA, ¬EA}	...	{PP, LRF, FRF, SA, ¬EA}	
PP ≺ LRF ≺ FRF ≺ SA ≺ EA	4	4	...	3	(4+4+...+3)/13
LRF ≺ PP ≺ FRF ≺ SA ≺ EA	4	4	...	4	(4+4+...+4)/13
⋮	⋮	⋮	⋮	⋮	⋮
LRF ≺ FRF ≺ SA ≺ EA ≺ PP	4	4	...	4	(4+4+...+4)/13

Table 2.2: Brute Force Approach to Compute the Optimal Ordering on Average

for the most trivial configuration models. To overcome the scalability limitations of the former approach, this paper proposes an heuristic solution grounded on the Information Theory concept of *entropy*.

CHAPTER

3

Related Work

Research on automated configurators is mainly focused on *consistency checking* and *optimization* [SW98, Jun06]. For example, reasoning engines such as BDD libraries, SAT solvers and Logic-Truth Maintenance systems have been used to detect invalid derivatives (i.e., those which violate some option dependency or incompatibility) [DGR11, SHN+07, Men09]; to provide explanations for configuration flaws [WBS+10, Jan10], to optimize configurations (i.e., to find configurations whose cost is less or equal than a given one) [HHRV11, SAH+11, SRK+11], etc.

Despite the importance of the interactive question ordering problem that our work tackles, which was pointed out by Steinberg more than thirty years ago [Ste80], there is little research on it. A recent approach that specifically deals with this problem is provided by Chen et al. [CE11], who propose to minimize the number of configuration steps by sorting the components according to their *probability*[1] of being included in a derivative. Such probability is computed by Equation 3.1.

$$\Pr(c) = \frac{\text{Number of valid derivatives that include } c}{\text{Total number of valid derivatives}} \tag{3.1}$$

In addition to Chen et al.'s approach, Mazo et al. [MDSD14] proposes the following heuristics for ordering configuration questions:

Heuristic 1 Components with the smallest domain first: choose first the component with the smallest domain. The domain of a component is the set of possible values that the component can take according to its domain definition and the constraints in which the component is involved.

[1]in their original paper and with a fully equivalent meaning, Chen et al. use the term *selectivity* instead of *probability*. As our approach follows an entropy driven heuristic and, the Information Theory concept of entropy is defined in terms of probability (see Section 4.1.1), we have preferred to use *probability* throughout this paper.

Heuristic 2 The most constrained components first: choose the component that participates in the largest number of constraints.

Heuristic 3 Components appearing in most products first. This heuristic is exactly the same as Chen et al's approach.

Heuristic 4 Automatic completion when there is no choice. This heuristic "provides a mechanism to automatically complete the configuration of variables where only one value of their domain is possible [...] it also works when a variable has several values on its domain but only one is valid". Strictly speaking, this is not a heuristic, but a propagation mechanism that all configuration systems should support. In remainder of this paper, we will assume that all heuristics include this mechanism.

Heuristic 5 Components required by the latest configured component first: choose the component that has the largest number of constraints with the past-configured components.

Heuristic 6 Components that split the problem space in two first: set first the components that divide the problem space in two parts of approximately the same size. Unfortunately, Mazo et al. do not provide a way to implement this heuristic which takes into account all model constraints. In particular, Mazo et al. propose a simplification by just using the tree structure of a FD, or the variation points of an orthogonal variability model [PBL05], but not processing the cross-tree constraints.

As it will be discussed in Section 4.2.4, our approach may be though as an implementation of Heuristic 6 that, in addition, takes into account all configuration model constraints. In Chapter 5 it will be shown that our approach provides better outcomes than Heuristics 1, 2, 3 (i.e., Chen's approach), and 5.

CHAPTER 4

Entropy-based approach to sort configuration questions

This chapter presents our heuristic to minimize the number of steps required to configure a derivative from a configuration model. Subsection 4.1.1 introduces the theoretical background of our approach. As we will see, our heuristic, as other ones summarized in Chapter 3, requires computing the component probabilities. Subsection 4.1.2 discusses the scalability limitations of the approach commonly used to compute those probabilities. To overcome such limitations, in Subsection 4.2 we propose an algorithm that provides an efficient probability computation. Finally, Subsection 4.2.4 describes our heuristic.

4.1 Preliminaries

4.1.1 Information Theory

The following definitions were originally introduced by Shannon [Sha48]. Let us start with the concept of *entropy*.

Let X be a discrete random variable with alphabet $\mathcal{X}$ and probability mass function $\Pr(x) = \Pr\{X = x\}$, $x \in \mathcal{X}$; the entropy H of X is defined by Equation 4.1:

$$H(X) = -\sum_{x \in \mathcal{X}} \Pr(x)\Big(\log_2 \Pr(x)\Big) \tag{4.1}$$

Let us present the concept of *conditional entropy*, which is the entropy of a random variable conditional on the knowledge of another random variable.

Let X and Y be two discrete random variables. The conditional entropy $H(X|Y)$ of X given Y is defined by Equation 4.2:

$$H(X|Y) = \sum_{y \in \mathcal{Y}} \Pr(y) H(X|Y = y) \tag{4.2}$$

Finally, let us introduce the concept of *mutual information*, also called *information gain*, which represents the reduction in a variable uncertainty due to another random variable.

Consider two random variables X and Y with a joint probability mass function $\Pr(x, y)$ and marginal probability mass functions $\Pr(x)$ and $\Pr(y)$. The mutual information $I(X; Y)$ is defined by Equation 4.3 as the relative entropy between the joint distribution and the product distribution $\Pr(x)\Pr(y)$:

$$I(X; Y) = \sum_{x,y} \Pr(x, y) \log_2 \frac{\Pr(x, y)}{\Pr(x)\Pr(y)} \tag{4.3}$$

Entropy and mutual information satisfy the following properties that will be used throughout this paper:

1. $H(X) \geq 0$

2. $H(X) \leq \log_2 \#\mathcal{X}$, with equality if and only if X is distributed uniformly over $\mathcal{X}$ (in this paper, the number of elements of a set S is denoted as $\#S$)

3. $I(X; Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) = I(Y; X)$

4.1.2 Straightforward approach to compute component probabilities

A widespread approach to support the automated management of configuration models is translating them to propositional logic formulas [SW98] [SHN+07], which are processed using off-the-self tools, such as SAT solvers [BHvM+09] or BDD engines [Bry86]. Table 4.1 summarizes the translations needed to encode our running example into propositional logic[1]. Equation 4.4 is the equivalent logic formula to Figure 2.1.

[1]a more detailed explanation on how to translate feature models into logic may be found in [TBK09].

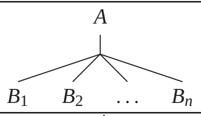
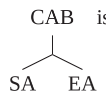
Type of relationship	Feature model representation	Translation to propositional logic
mandatory	$A \multimap\!\!\bullet\ B$	$(\neg A \lor B) \land (\neg B \lor A)$
optional	$A \multimap \mathrm{B}$	$\neg B \lor A$
alternative		$\bigwedge_{i=1}^{n}(\neg B_i \lor A) \land (\neg A \bigvee_{i=1}^{n} B_i) \bigwedge_{i<j}(\neg B_i \lor \neg B_j)$
requires	$A \xrightarrow{requires} B$	$\neg A \lor B$

Table 4.1: Equivalence between configuration models and propositional logic formulaes

$C \multimap \mathrm{ADC}$is translated to $(\neg\mathrm{ADC} \lor C)\land$

$\mathrm{ADC} \multimap\!\!\bullet\ \mathrm{CAB}$is translated to $(\neg\mathrm{ADC} \lor \mathrm{CAB}) \land (\neg\mathrm{CAB} \lor \mathrm{ADC})\land$

 is translated to $(\neg\mathrm{SA} \lor \mathrm{CAB}) \land (\neg\mathrm{EA} \lor \mathrm{CAB}) \land (\neg\mathrm{CAB} \lor \mathrm{SA} \lor \mathrm{EA}) \land (\neg\mathrm{SA} \lor \neg\mathrm{EA})\land$

$\mathrm{ADC} \multimap \mathrm{PP}$is translated to $(\neg\mathrm{PP} \lor \mathrm{ADC})\land$

$C \multimap\!\!\bullet\ S$is translated to $(\neg\mathrm{S} \lor \mathrm{C}) \land (\neg\mathrm{C} \lor \mathrm{S})\land$

$S \multimap \mathrm{LRF}$is translated to $(\neg\mathrm{LRF} \lor S)\land$

$S \multimap \mathrm{FRF}$is translated to $(\neg\mathrm{FRF} \lor S)\land$

$\mathrm{PP} \xrightarrow{requires} \mathrm{LRF}$is translated to $(\neg\mathrm{PP} \lor \mathrm{FRF})$

(4.4)

Once a configuration model is encoded into a logic formula ψ:

- the total number n_1 of valid derivatives is equivalent to the number of satisfying assignments of ψ (i.e., those that evaluates ψ to true).
- the number n_2 of valid derivatives that include component c is equivalent to the number of satisfying assignments of $\psi \land c$.

Since the probability of a component is $\frac{n_2}{n_1}$, and the computation of the number of satisfying assignments of a Boolean formula is supported by most BDD engines and SAT solvers (in particular, #SAT counters are a type of SAT solvers specifically oriented to compute such number), a straightforward approach to compute the component probabilities is calling repeatedly a logic engine using $\psi \land c_i$ as input [KZK10]. Unfortunately, this approach has an high computational cost and does not scale for all but trivial configuration models. While the SAT problem is known to be NP-complete [Coo71], it is widely believed that the #SAT problem is even harder [BHvM+09]. If n is the number of components, computing the component probabilities requires calling a #SAT solver n times, which is extremely time-consuming. Similarly, computing the number of satisfying assignments with a BDD has computational complexity $O(m)$ [Bry86], where m is the number of nodes of the BDD. Hence, the complexity of computing the component probabilities by calling repeatedly the BDD engine is $O(n \cdot m)$, which is excessively time-consuming for most configuration models.

For instance, it is well known by the car manufacturing community that the first issue of car configurators is performance [ACF10]. Thus, as soon as customers make a configuration choice they want to find out what the consequences of the choice are. From a marketing perspective, it is unpleasant for customers to wait for several seconds to know whether their requirements are correct or not in terms of configuration. As it will be shown experimentally in Chapter 5, computing the component probabilities by calling repeatedly a BDD may force the costumer to wait for more than 600 seconds for just a single configuration step!

To overcome the aforementioned scalability limitations, in the following section we propose a BDD algorithm that computes component probabilities in almost linear time to m.

4.2 Efficient computation of the probabilities of the variables of a Boolean formula from a BDD

BDDs are a way of representing Boolean functions. They are rooted, directed, acyclic graphs, which consist of several decision nodes and terminal nodes [Bry86]. There are two types of terminal nodes called 0-terminal and 1-terminal. Each decision node v_i is labeled by a Boolean variable x_k and has two child nodes called *low* and *high* (which are usually depicted by dashed and solid lines, respectively). The edge from node v_i to a low (or high) child represents an assignment of v_i to 0 (resp. 1). Such a BDD is called *ordered* if different variables appear in the same order on all paths from the root. A BDD is said to be *reduced* if the following two rules have been applied to its graph: (i) isomorphic subgraphs are merged, and (ii) nodes whose two children are isomorphic are eliminated.

Let us use formula $\psi \equiv (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ as running example for this subsection. Table 4.2 is the truth table for ψ. Figure 4.1 is its BDD[1] representation using the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$[2].

The remainder of this subsection is structured as follows. Firstly, some definitions required to understand our algorithm are given. Next, the data structures the algorithm uses are described from a theoretical perspective. Then, the algorithm is presented. Finally, the algorithm computational cost is discussed.

[1] in popular usage, the term BDD almost always refers to Reduced Ordered Binary Decision Diagram [HR04]. In this paper, we will follow that convention as well

[2] note that a logic formula may be encoded with different BDDs according to the variable ordering used to synthesize the BDD. Obviously, our algorithm produces the same results for equivalent BDDs (i.e., BDDs that encode the same formula)

x_1	x_2	x_3	x_4	ψ
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 4.2: Truth table for $\psi \equiv (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

4.2.1 Definitions

The satisfying set of a Boolean formula $\psi(x_1, ..., x_n)$, denoted S_ψ, is defined by Equation 4.5.

$$S_\psi = \{(x_1, ..., x_n) | \psi(x_1, ..., x_n) = \text{true}\} \tag{4.5}$$

The satisfying set of the variable x_i of a Boolean formula $\psi(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n)$, denoted $S_{\psi|_{x_i=\text{true}}}$, is defined by Equation 4.6.

$$S_{\psi|_{x_i=\text{true}}} = \{(x_1, ..., x_{i-1}, \text{true}, x_{i+1}, ..., x_n) | \psi(x_1, ..., x_{i-1}, \text{true}, x_{i+1}, ..., x_n) = \text{true}\} \tag{4.6}$$

For instance, according to Table 4.2, $\#S_\psi = 7$ since there are 7 rows where ψ evaluates to true[1], and $\#S_{\psi|x_4} = 5$ because $x_4 = 1$ in 5 of the 7 rows where $\psi = 1$.

The satisfying probability of a Boolean formula $\psi(x_1, ..., x_n)$, denoted $\text{Pr}(\psi)$, is defined by Equation 4.7.

$$\text{Pr}(\psi) = \frac{\#S_\psi}{2^n} \tag{4.7}$$

The satisfying marginal probability of a variable x_i in a Boolean formula $\psi(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n)$, denoted $\text{MPr}(\psi|_{x_i=\text{true}})$, is defined by Equation 4.8.

$$\text{MPr}(\psi|_{x_i=\text{true}}) = \frac{\#S_{\psi|_{x_i=\text{true}}}}{2^n} \tag{4.8}$$

[1]throughout this paper 0/1 and false/true are used interchangeably

The satisfying probability of a variable x_i in a Boolean formula $\psi(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n)$, denoted $\Pr(\psi|_{x_i=\text{true}})$, is defined by Equation 4.9.

$$\Pr(\psi|_{x_i=\text{true}}) = \frac{\#S_{\psi|_{x_i=\text{true}}}}{\#S_{\psi}} \tag{4.9}$$

For instance, looking at Table 4.2, it is easy to see that $\Pr(\psi) = \frac{7}{2^4}$, $\text{MPr}(\psi|_{x_4=\text{true}}) = \frac{5}{2^4}$, and $\Pr(\psi|_{x_4=\text{true}}) = \frac{5}{7}$. For convenience, in the remainder of the paper we denote $\Pr(\psi|_{x_i=\text{true}})$ and $\text{MPr}(\psi|_{x_i=\text{true}})$ as $\Pr(x_i)$ and $\text{MPr}(x_i)$, respectively.

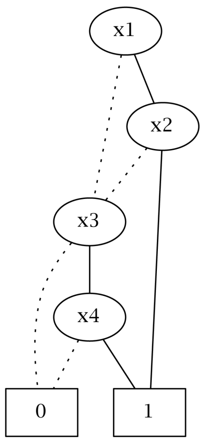


Figure 4.1: BDD for ψ according to the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$

4.2.2 Data Structures

Let us represent a BDD with has m nodes and encodes a Boolean formula with n variables by using the following data structures:

- The variable ordering used to synthesize the BDD is represented by an array declared as follows:

```
var_ordering: array[0..n-1] of string
```

- Each node is represented by a record declared as follows:

```
type node = record
    index: 0..n
    low, high: node
    mark: Boolean
end
```

Where:

1. *index* is the index of the variables in the ordering. The terminal nodes of the BDD (i.e., 0 and 1) have index *n*.
2. *low* and *high* are the low and high node successors
3. *mark* is used to mark which nodes have been visited during a traversal of the graph. As we will see, our algorithm is called at the top level with the root node as argument and with the mark fields of the nodes being either all true or all false. It then systematically visits every node in the graph by recursively visiting the subgraphs rooted by the two children *low* and *high*. As it visits a node, it complements the value of the *mark* field, so that it can later determine whether a child has already been visited by comparing the two marks.

- The BDD is represented by an array declared as follows:

```
bdd: array[0..m] of node
```

The terminal nodes of the BDD, 0 and 1, are stored at positions 0 and 1 of the *bdd* array, respectively.

For instance, Tables 4.3 and 4.4 represent the content of *bdd* and *var_ordering* for the BDD in Figure 4.1, respectively.

position	index	low	high	mark
0	4	nil	nil	false
1	4	nil	nil	false
2	3	0	1	false
3	2	0	2	false
4	1	3	1	false
5	0	3	4	false

Table 4.3: Content of the *bdd* array for Figure 4.1

position	content
0	“x_1”
1	“x_2”
2	“x_3”
3	“x_4”

Table 4.4: Content of the *var_ordering* array for the Figure 4.1

4.2.3 Algorithm

$\Pr(x_i)$ is computed jointly by Algorithms 1, 2 and 3. Figure 4.2 summarizes the computations for the BDD in Figure 4.1. Let us examine how our approach proceeds:

Algorithm 1 computes $\Pr(x_i)$ as $\Pr(x_i) = \frac{\mathrm{MPr}(x_i)}{\Pr(\psi)}$ by calling the auxiliary Algorithms 2 and 3.

Algorithm 2 computes $\Pr(\psi)$. A nice mental picture to understand Algorithm 2 is thinking in pouring 1 liter of water from the BDD root to the terminal nodes. 1 liter goes through the root, then half a liter goes through the low branch and half a liter through the high branch. This procedure advances until the water reaches the leaves. Hence, $\mathrm{MPr}(x_i)$ is the amount of water that node 1 has.

In Figure 4.1, through node v_5 goes 1 liter (i.e., formula_sat_prob[5][1] = 1). Half of it goes to v_3 and the other half to v_4. Whereas through v_4 passes $\frac{1}{2}$ liter, through v_3 goes the $\frac{1}{2}$ liter that comes from v_5 and half of the water that comes from v_4 (i.e., formula_sat_prob[3] = $\frac{1}{2} + \frac{\frac{1}{2}}{2} = \frac{3}{4}$).

Algorithm 3 computes $\mathrm{MPr}(x_i)$. In particular, let us examine how it computes $\mathrm{MPr}(x_2)$. In the truth Table 4.2, ψ evaluates to true when x_2 is true five times:

1. In four of them x_1 is true. When the call get_marginal_prob(4, ...) is made, lines 10-23 compute the marginal probability of x_2 for the explicit path $v_5 \rightarrow v_4$. The probabilities due to the low and high branches of v_i are stored into the prob_low and prob_high variables, respectively. As bdd[4].low $\neq 0$, a recursive call is made to compute the total probability due to the low descendants of v_4 (i.e., get_marginal_prob(3, ...)). As a result:

$$\text{total\_prob}[3] = \text{prob\_low}_{v_3} + \text{prob\_high}_{v_3} = 0 + \frac{3}{16} = \frac{3}{16}$$

   Notice that prob_low$_{x_2}$ is not simply equal to total_prob[3], because total_prob[3] depends also on the probability that comes from the link $v_5 \dashrightarrow v_3$. To get just the probability due to the link $v_4 \dashrightarrow v_3$, prob_low has to be adjusted using the formula_sat_prob array as:

$$\text{prob\_low} = \frac{\text{total\_prob}[3] \cdot \frac{\text{formula\_sat\_prob}[4]}{2}}{\text{formula\_sat\_prob}[3]} = \frac{\frac{3}{16} \cdot \frac{\frac{1}{2}}{2}}{\frac{3}{4}} = \frac{1}{16}$$

   Since bdd[4].high = 1, prob_high is directly computed as:

$$\text{prob\_high} = \frac{\text{formula\_sat\_prob}[4]}{2} = \frac{\frac{1}{2}}{2} = \frac{1}{4}$$

[1] according to Tables 4.3 and 4.4, the root node has label v_5 and it is in the position 5 of the *bdd* array

Finally:

$$\text{prob[bdd[4].index]} = \text{prob\_high} = \frac{1}{4}$$

2. In one of them x_1 is false. The two following implicit paths that have been removed from the reduced BDD: (i) $v_5 \dashrightarrow v_4 \dashrightarrow v_3$, and (ii) $v_5 \dashrightarrow v_4 \rightarrow v_3$. Nevertheless, path $v_5 \dashrightarrow v_4 \rightarrow v_3$ should be considered to compute the marginal probability of x_2. Lines 24-31 account for that kind of implicit paths, adjusting the marginal probability with the variables omitted in the paths. For instance, when the algorithm is called for v_5, the marginal probability of x_2 is updated with half the prob_low of v_5.

To sum up:

$$\text{MPr}(x_2) = \text{MPr}(v_5 \dashrightarrow v_4 \rightarrow v_3) + \text{MPr}(v_5 \rightarrow v_4) = \frac{\text{prob\_low}_{v_5}}{2} + \text{prob[bdd[4].index]} = \frac{\frac{1}{8}}{2} + \frac{1}{4} = \frac{5}{16}$$

Algorithm 1: get_prob

```
Input bdd and var_ordering arrays;
Output an array which stores $\Pr(x_i)$ in position i ;
var formula_sat_prob, total_prob: array[0..length(bdd)-1] of float;;
    prob: array[0..length(var_ordering)-1] of float; i: int;;
begin
    for (i=0; i < length(bdd); i++) do
        total_prob[i] = 0.0;
    for (i=0; i < length(var_ordering); i++) do
        prob[i] = 0.0;
    formula_sat_prob = get_formula_sat_prob(bdd);
    get_marginal_prob(length(bdd)-1, total_prob, formula_sat_prob, prob, bdd, var_ordering);
    for (i=0; i < length(var_ordering); i++) do
        prob[i] = $\frac{\text{prob[i]}}{\text{formula\_sat\_prob[1]}}$;
    return prob
end
```

4.2.3.1 **Computational cost**

Let m be the number of nodes of the BDD, and n the number of variables of the Boolean formula. Algorithm 2 requires traverse all the nodes, so its computational complexity is $O(m)$. Algorithm 3 also traverses all the BDD nodes. In addition, to account for the implicit paths removed from the reduced BDD, the variables omitted on the edges that come from each node need to be traversed (which is done by lines 24-31). Table 4.5 summarizes those traversals for Figure 4.1. For instance, when v_4 is recursively traversed, the variables x_3 and x_4 need to be iteratively

Algorithm 2: get_formula_sat_prob

```
1  Input bdd array;
2  Output an array which in position 1 stores $Pr(\psi)$;
3  var formula_sat_prob: array[0..length(bdd)-1] of float; i: int;
4  begin
5      for (i=0; i < length(bdd)-1; i++) do
6          formula_sat_prob[i] = 0.0 // non-root nodes prob is initialized to 0
7      formula_sat_prob[i] = 1.0 // root node prob is 1
8      i=length(bdd)-1;
9      while $i > 1$ do // for all non-terminal nodes
10         formula_sat_prob[bdd[i].low] += $\frac{formula\_sat\_prob[i]}{2.0}$;
11         formula_sat_prob[bdd[i].high] += $\frac{formula\_sat\_prob[i]}{2.0}$;
12         i -= 1
13     return formula_sat_prob
14 end
```

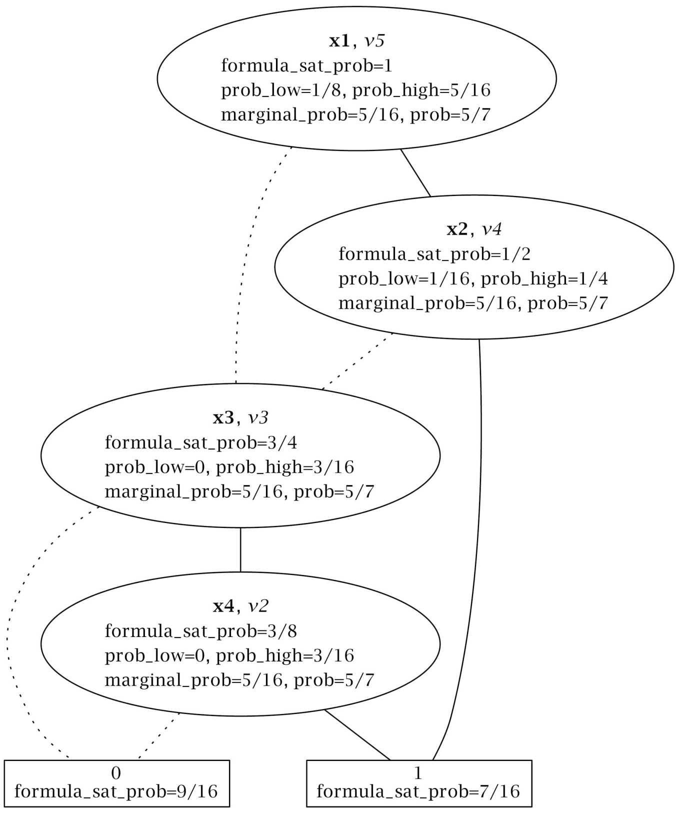

x1, v5
formula_sat_prob=1
prob_low=1/8, prob_high=5/16
marginal_prob=5/16, prob=5/7
x2, v4
formula_sat_prob=1/2
prob_low=1/16, prob_high=1/4
marginal_prob=5/16, prob=5/7
x3, v3
formula_sat_prob=3/4
prob_low=0, prob_high=3/16
marginal_prob=5/16, prob=5/7
x4, v2
formula_sat_prob=3/8
prob_low=0, prob_high=3/16
marginal_prob=5/16, prob=5/7
0
formula_sat_prob=9/16
1
formula_sat_prob=7/16

Figure 4.2: Probability computation for BDD 4.1

Algorithm 3: get_marginal_prob

```
Input v: 0..length(bdd)-1; total_prob, formula_sat_prob: array[0..length(bdd)-1] of float;;
    prob: array[0..length(var_ordering)-1] of float; bdd and var_ordering arrays;
Output prob is passed by reference and, at the end of the algorithm execution,;
    it stores MPr($x_i$) in position i ;
var prob_low, prob_high: float; i: int;
begin
    prob_low = 0.0;
    prob_high = 0.0;
    bdd[v].mark = not bdd[v].mark;
    // explicit path recusive traversal
    if bdd[v].low == 1 then
        prob_low = $\frac{\text{formula\_sat\_prob}[v]}{2.0}$;
    else if bdd[v].low ≠ 0 then
        if bdd[v].mark ≠ bdd[bdd[v].low].mark then
            get_marginal_prob(bdd[v].low, total_prob, formula_sat_prob, prob, bdd, var_ordering);
        prob_low = $\frac{\text{total\_prob}[\text{bdd}[v].\text{low}] \cdot \frac{\text{formula\_sat\_prob}[v]}{2.0}}{\text{formula\_sat\_prob}[\text{bdd}[v].\text{low}]}$
    if bdd[v].high == 1 then
        prob_high = $\frac{\text{formula\_sat\_prob}[v]}{2.0}$;
    else if bdd[v].high ≠ 0 then
        if bdd[v].mark ≠ bdd[bdd[v].high].mark then
            get_marginal_prob(bdd[v].high, total_prob, formula_sat_prob, prob, bdd, var_ordering);
        prob_high = $\frac{\text{total\_prob}[\text{bdd}[v].\text{high}] \cdot \frac{\text{formula\_sat\_prob}[v]}{2.0}}{\text{formula\_sat\_prob}[\text{bdd}[v].\text{high}]}$
    total_prob[v] = prob_low + prob_high;
    prob[bdd[v].index] += prob_high;
    // implicit path iterative traversal
    i = bdd[v].index + 1;
    while i<bdd[bdd[v].low].index do
        prob[i] += $\frac{\text{prob\_low}}{2.0}$;
        i +=1
    i = bdd[v].index + 1;
    while i<bdd[bdd[v].high].index do
        prob[i] += $\frac{\text{prob\_high}}{2.0}$;
        i +=1
end
```

traversed because the edge $v_4 \rightarrow 1$ omits them (i.e., the variable encoded by node v_4, x_2, jumps directly to 1 omitting the intermediate variables x_3 and x_4 in the ordering $x_1 \prec x_2 \prec x_3 \prec x_4$). Table 4.5 helps noticing the savings our algorithm provides compared to the straightforward approach described in Section 4.1.2, which requires traversing all nodes for all variables (which in computational cost terms is equivalent to traversing all variables for every node). Therefore, Algorithm 3 does not traverse $m \cdot n$ elements, but $m \cdot n'$, where n' is strictly less than n[1].

It follows that Algorithm 1 has computational complexity $O(m \cdot n')$. As it will be shown in Section 5, in the practice n' is usually much smaller than n and thus variable probabilities can be efficiently computed.

node	**arcs**	**omitted vars that are traversed**
v_5	$v_5 \dashrightarrow v_3$	x_2
	$v_5 \rightarrow v_4$	none
v_4	$v_4 \dashrightarrow v_3$	none
	$v_4 \rightarrow 1$	x_3, x_4
v_3	$v_3 \dashrightarrow 0$	x_4
	$v_3 \rightarrow v_2$	none
v_5	$v_5 \dashrightarrow 0$	none
	$v_5 \rightarrow 1$	none

Table 4.5: Variables iteratively traversed for BBD in Figure 4.1

[1]If $n' = n$, all nodes in the BDD should go directly to 0 or 1, jumping over all the variables. Nevertheless, as BDDs are organized in hierarchical levels according to the variable ordering, this is impossible (i.e., the nodes that encode a variable with position k in the ordering only can jump over the variables with positions $k + 1 \ldots n$)

4.2.4 Entropy driven configuration

Let us return to the original problem this paper tackles. Given a set of questions $\mathcal{Q}$, our goal is to sort it in such a way that the user has to answer as few questions as possible to complete the configuration. To find the optimal order of $\mathcal{Q}$, we propose to rank each question q according to its expected information gain, i. e., measuring how much uncertainty can be reduced on average when the engineer answers it. Such information gain is modeled as the mutual information $I(\mathcal{C}; q)$, where $\mathcal{C}$ is the set of all valid configurations (i. e., the ones that satisfy all asset interdependencies).

When a configuration is completed, the entropy of every question q is zero. Since q has been answered, $H(q|\mathcal{C}) = 0$. Thus, it follows that $I(\mathcal{C}; q) = H(q)$, as Equation 4.10 demonstrates (see Property 3 in Subsection 4.1.1).

$$I(\mathcal{C}; q) = H(q) - H(q|\mathcal{C}) = H(q) \tag{4.10}$$

When we ask "is component c in the configuration?", the entropy of the question $H(q)$ is computed by Equation 4.11, where $\Pr(c)$ is the probability that c is included in the configuration.

$$\begin{aligned} H(q) &= -\Pr(c)\log_2\Pr(c) - \Pr(\neg c)\log_2\Pr(\neg c) \\ &= -\Pr(c)\log_2\Pr(c) - (1 - \Pr(c))\log_2(1 - \Pr(c)) \end{aligned} \tag{4.11}$$

Our approach to guide the configuration of a derivative may be thought of as a binary search for the user desired configuration (remember Heuristic 6 in Chapter 3). To successively divide the search space into subspaces of approximately the same size (i. e., where the pursued configuration is approximately with the same probability), the user answers the question that provides more information about the configuration (i. e., the question with highest entropy). Thus, the configuration process advances iteratively, by performing the following activities, until the entropy of all components becomes zero:

1. Computing the component probabilities from the input configuration model. As the process advances, the configuration space gets narrower and, consequently, the component probabilities change.

2. Computing the entropy value for each question.

3. Sorting the questions in descending order of entropy.

4. Asking the user for answering a question with entropy greater than zero. Note than when a question has zero entropy, it is because it has been answered in a previous step directly or indirectly (i. e., because of the question interdependencies).

5. Updating the set of answers and the configuration model[1].

[1] e.g., if the customer answers negatively a question q, the Boolean formula ψ that encodes the configuration model is updated to $\psi \wedge \neg f$

Entropy may also be used to measure how hard is to configure a given model. From the "point of view" of an automated configurator, when the configuration process starts the derivative desired by the customer is any c in $\mathcal{C}$ with the same probability. So the configuration model uncertainty is calculated by Equation 4.12 (see Property 2 in Section 4.1.1).

$$H(\mathcal{C}) = \log_2 \#\mathcal{C} \tag{4.12}$$

4.2.4.1 **Example**

Coming back to the running example introduced in Chapter 2, let us see how our approach works. Figure 4.3 sums up the steps required to configure the derivative {PP, LRF, ¬FRF, SA, ¬EA} using the entropy heuristic. In the first step, EA is the component with highest entropy. So the system asks the user if SA is included in the derivative. Once the user answers affirmatively, the probabilities of the components are recomputed and so the entropies (e.g., the inclusion of SA implies the exclusion of EA, so Pr(EA)=0 and thus H(EA)=0).

Component	H
PP	0.78
LRF	0.96
FRF	0.96
SA	0.99
EA	0.78

SA ? Yes

$\rightarrow$

Component	H
PP	0.92
LRF	0.92
FRF	1
SA	0
EA	0

FRF ? No

$\rightarrow$

Component	H
PP	0.92
LRF	0.92
FRF	0
SA	0
EA	0

PP ? Yes

Figure 4.3: Configuring derivative {PP, LRF, ¬FRF, SA, ¬EA} using component entropy

We remark here that our approach does not force the user to follow a fixed sequence of questions. In each configuration step, the user may decide not to answer the best entropy-ranked question, but the one she thinks is more convenient. After the question is answered, the entropies are recomputed and thus our approach adjust to the user preferences in an interactive way.

CHAPTER
5

Experimental evaluation

To test the validity of our approach, we have used two case studies:

1. The configuration model provided by the car manufacturing company Renault DVI[1], which deals with the configuration of a family of cars named Renault Megane[2]. We have selected this model because it illustrates the practical applicability of our approach (i.e., instead of using an example made up for academic purposes, our work is tested on a real configuration model that comes from the industry). In addition, the Renault Megane problem is a benchmark of widespread use by the configuration community [AFM02] [Jen04] [OOF05] [HHOW05] [NW07] [HT07] [CO08] [Que11] [Kro12] [Gan12] [BFL13].

2. The Electronic Shopping model provided by Lau [Lau06], which deals with an electronic commerce platform that allows consumers to directly buy goods or services from a seller over the Internet using a web browser[3]. This benchmark is widely used by the software product line community [Men09] [BG11] [POS$^+$12].

5.1 Experimental design

The goal of this section is to check if:

- Our approach produces better results than related work.

[1] `http://www.renault.fr/`
[2] The Renault Megane configuration model is freely available at `http://www.itu.dk/research/cla/externals/clib/`
[3] The Electronic Shopping model is freely available at `http://gsd.uwaterloo.ca:8088/SPLOT/feature_model_repository_de`

- The algorithm presented in Section 4.2 provides reasonable response times and thus support customer interactivity during the configuration process.

To do so, we have created a test bed composed of 1000 random derivatives for every configuration model. As we will see, a sample of 1000 derivatives is big enough to get results with high statistical power and significance.

To generate valid derivatives that satisfy all constraints, we have encoded the models as propositional logic formulaes (see Subsection 4.1.2) and then as BDDs. To get efficient BDD sizes, the directions given by Narodytska et al. [NW07] have been followed. The BuDDy package[1] has been used to guarantee the generation of valid derivatives (i.e., derivatives that conform to the BDD).

The test bed is used to compare the following methods:

1. Mazo et al.'s Heuristics 1, 2 and 5[2].

2. *Probability* driven approach, i.e., the method proposed by Chen et al. and Mazo et al. (Heuristic 3).

3. *Entropy* driven approach, i.e., the method we propose in this paper.

To compute the option probabilities, which are required by the *entropy* and *probability* approaches, an implementation of the algorithm presented in Section 4.2 has been included into the BuDDy package.

5.1.1 Case study 1: Renault Megane

5.1.1.1 Results

Table 5.1 summarizes the results of the experiments for the Renault Megane configuration model. Histograms in Figure 5.1.a represent the number of steps needed to configure the 1000 derivatives using Mazo et al.'s Heuristics 1, 2 and 5, and the *entropy* and *probability* approaches. Figure 5.1.b complements the histogram representation of the results with a box plot[3].

Using the Central Limit Theorem, the 95% Confidence Intervals (CI) of the population mean can be estimated (i.e., the range where, with a 95% guarantee, the mean of the number of steps required to configure every derivative of the Megane model lies). Table 5.2 summarizes the CIs for each approach[4].

[1]BuDDy is freely available at `http://sourceforge.net/projects/buddy/`

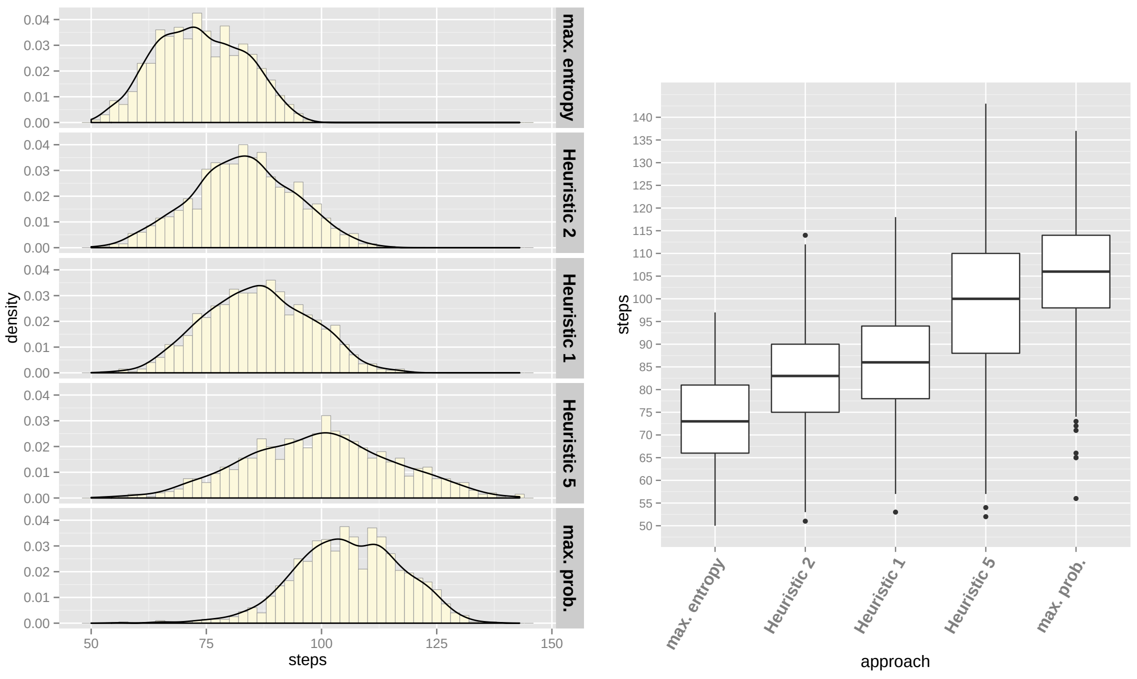
[2]remember that, strictly speaking, Mazo et al.'s Heuristic 4 is not a *heuristic*, but a propagation mechanism that all configuration systems should support. So we have included such mechanism in all the methods tested in this paper.

[3]"whiskers" in Figure 5.1.b start from the edge of the box and extend to the furthest data point that is within 1.5 times the inter-quartile range (i.e., the range that goes from the 25th percentile to the 75th percentile). Points that are past the ends of the whiskers have been considered outliers and displayed with dots.

[4]CIs are estimated as $\boxed{\text{population mean CI} = \text{sample mean} \pm t(\text{std. error}, 95\%, 999 \text{ degrees of freedom})}$, where t stands for the Student's t-distribution.

approach	mean	std. deviation	median	min	max	range
entropy	73.49	9.5	73	50	97	47
probability	105.79	11.54	106	56	137	81
Heuristic 1	86.04	11.26	86	53	118	65
Heuristic 2	82.74	11.12	83	51	114	63
Heuristic 5	99.39	15.95	100	52	143	91

Table 5.1: Result of the experiments for Renault Megane



(a) Histograms (b) Box plots

Figure 5.1: Number of configuration steps according to the used approach for Renault Megane

entropy		probability		Heuristic 1		Heuristic 2		Heuristic 5	
std. error	**95% CI**	**std. error**	**95% CI**	**std. error**	**95% CI**	**std. error**	**95% CI**	**std. error**	**95% CI**
0.3	72.90-74.08	0.36	105.07-106.50	0.36	85.35-86.74	0.35	82.05-83.43	0.5	98.40-100.38

Table 5.2: 95% CI of the population mean for Renault Megane

According to the summarized data, there is experimental evidence supporting that our approach produces better results than related work.

5.1.1.2 Statistical significance

To check the statistical significance of the results, an Analysis of Variance (ANOVA) test has been run on the experimental data. Table 5.3 summarizes the ANOVA outcomes. Since the p-value is less than 0.001 (in particular, p-value $< 2 \cdot 10^{-16}$), the experimental results are statistically highly significant.

	degrees of freedom	sum of squares	mean of squares	F-value	Pr($> F$)
approaches	4	676884	169221	1162	$< 2 \cdot 10^{-16}$
residuals	4995	727312	146		

Table 5.3: ANOVA test for Renault Megane

Table 5.4 summarizes the power analysis of the ANOVA test. Given the sample size and the high effect size (i.e., the high values of η^2 and Cohen's f^2), the ANOVA test has high statistical power.

effect size		power
eta squared η^2	**Cohen's f^2**	
0.48	0.93	≈ 1

Table 5.4: Power analysis for Renault Megane

Finally, to check the statistical significance of the pairwise comparison between the approaches, a Tukey Honest Significant Differences (HSD) has been run. According to the results, summarized in Table 5.5, the difference between the number of steps required by any pair of approaches to configure a derivative is statistically highly significant[1].

5.1.2 Case study 2: Electronic Shopping

Table 5.6 and Figure 5.2 summarize the results of the experiments for the Electronic Shopping configuration model. Table 5.7 summarizes the CIs for each approach. According to the outcomes, there is experimental evidence supporting that our approach produces better results than related work.

[1]whereas the ANOVA test rejects the null hypothesis: "there is no difference between the five approaches (i.e., all of them produce approximately the same results)", Tukey HSD test rejects ten null hypotheses separately: "there is no difference between the Heuristic 2 and the entropy approach", "there is no difference between Heuristic 1 and the entropy approach ", etc.

	difference	**95% CI**	**adjusted p-value**
Heuristic 2 *vs* entropy	9.25	7.78-10.72	≈ 0
Heuristic 1 *vs* entropy	12.56	11.08-14.02	≈ 0
Heuristic 5 *vs* entropy	25.89	24.42-27.37	≈ 0
probability *vs* entropy	32.29	30.82-33.77	≈ 0
Heuristic 1 *vs* Heuristic 2	3.30	1.83-4.77	≈ 0
Heuristic 5 *vs* Heuristic 2	16.65	15.17-18.12	≈ 0
probability *vs* Heuristic 2	23.04	21.57-24.52	≈ 0
Heuristic 5 *vs* Heuristic 1	13.34	11.87-14.82	≈ 0
probability *vs* Heuristic 1	19.74	18.27-21.22	≈ 0
probability *vs* Heuristic 5	6.40	4.93-7.87	≈ 0

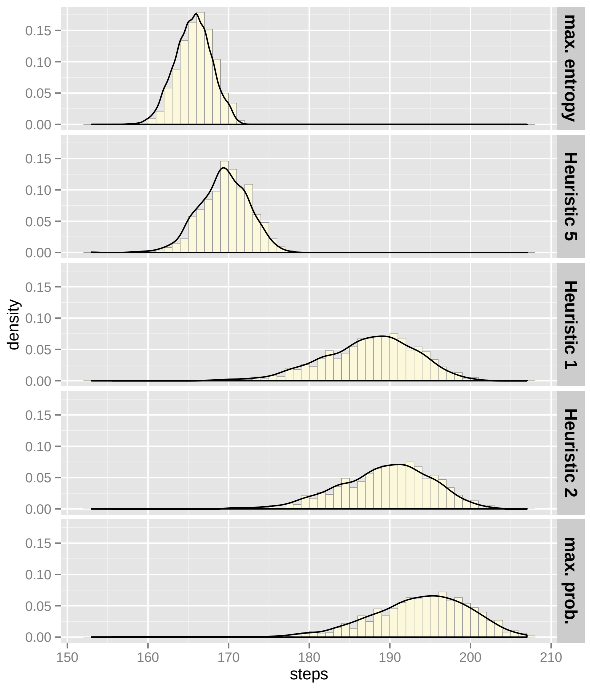
Table 5.5: Tukey HSD test for Renault Megane

approach	**mean**	**std. deviation**	**median**	**min**	**max**	**range**
entropy	165.57	2.23	166	158	171	13
probability	193.67	6.06	194	164	207	43
Heuristic 1	187.38	5.69	188	168	201	33
Heuristic 2	189.36	5.7	190	170	203	33
Heuristic 5	169.33	3.1	169	153	178	25

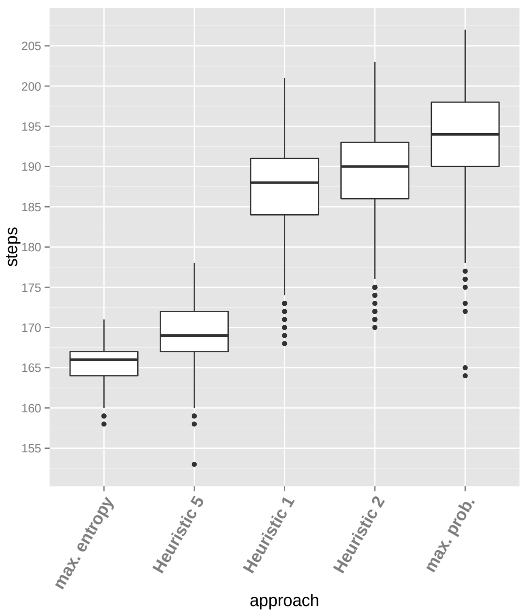
Table 5.6: Result of the experiments for Electronic Shopping

entropy		**probability**		**Heuristic 1**		**Heuristic 2**		**Heuristic 5**	
std. error	**95% CI**	**std. error**	**95% CI**	**std. error**	**95% CI**	**std. error**	**95% CI**	**std. error**	**95% CI**
0.07	165.43-165.71	0.19	193.29-194.04	0.18	187.03-187.73	0.18	189.01-189.72	0.1	169.14-169.52

Table 5.7: 95% CI of the population mean for Electronic Shopping



(a) Histograms



(b) Box plots

Figure 5.2: Number of configuration steps according to the used approach for Electronic Shopping

5.1.2.1 Statistical significance

Table 5.8 summarizes the ANOVA outcomes. Since the p-value is less than 0.001 (in particular, p-value $< 2 \cdot 10^{-16}$), the experimental results are statistically highly significant. Table 5.9 summarizes the power analysis of the ANOVA test. Given the sample size and the high effect size, the ANOVA test has high statistical power. Finally, Table 5.10 summarizes the outcomes of HSD, which show that the difference between the number of steps required by any pair of approaches to configure a derivative is statistically highly significant.

	degrees of freedom	**sum of squares**	**mean of squares**	**F-value**	**Pr($> F$)**
approaches	4	645314	161328	6950	$< 2 \cdot 10^{-16}$
residuals	4995	115944	23		

Table 5.8: ANOVA test for Electronic Shopping

effect size		**power**
eta squared η^2	**Cohen's f^2**	
0.85	5.57	≈ 1

Table 5.9: Power analysis for Electronic Shopping

	difference	**95% CI**	**adjusted p-value**
Heuristic 5 *vs* entropy	3.76	3.17-4.35	≈ 0
Heuristic 1 *vs* entropy	21.81	21.22-22.40	≈ 0
Heuristic 2 *vs* entropy	23.79	23.21-24.38	≈ 0
probability *vs* entropy	28.09	27.51-28.68	≈ 0
Heuristic 1 *vs* Heuristic 5	18.05	17.46-18.64	≈ 0
Heuristic 2 *vs* Heuristic 5	20.03	19.45-20.62	≈ 0
probability *vs* Heuristic 5	24.33	23.75-24.92	≈ 0
Heuristic 2 *vs* Heuristic 1	1.98	1.39-2.57	≈ 0
probability *vs* Heuristic 1	6.28	5.69-6.87	≈ 0
probability *vs* Heuristic 2	4.30	3.71-4.89	≈ 0

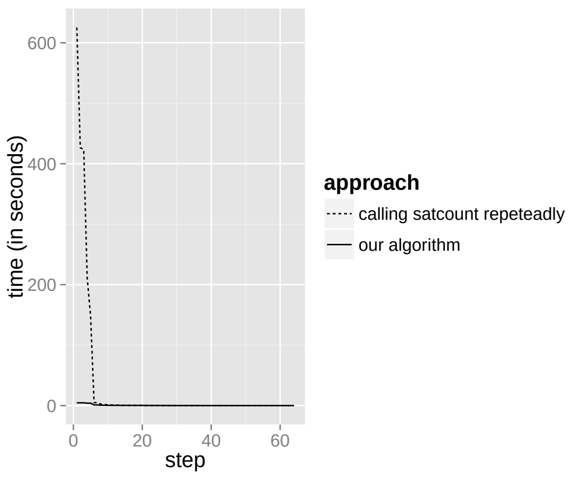
Table 5.10: Tukey HSD test for Electronic Shopping

5.1.3 Threats to Validity

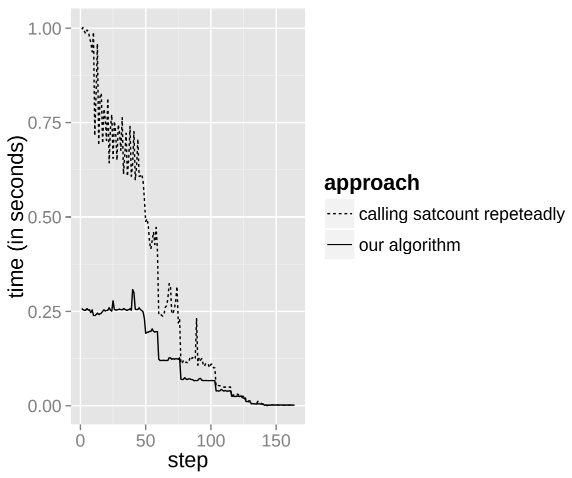
A threat to the validity of our approach is the time required to compute the component probabilities[1]. For the sake of interactivity, configurators must provide customer guidance in a short-time and the usual way to compute the probabilities is highly time-consuming (see Section 4.1.2). To assess the response time of our algorithm (see Section 4.2), we determined the time needed to configure 1000 randomly generated derivatives for the case studies 1 and 2 using our entropy-driven approach. Figure 5.3 compares the average times needed to completely configure the derivatives by computing the component probabilities using our algorithm, and calling repeatedly the BuDDy function *satcount*. The performance tests were conducted on an Intel© Core^{TM} 2 i3-4010U with 1.7 GHz and 4GB RAM (although only one core was used).

As Figure 5.3 shows, our algorithm greatly improves the probability computation time. For instance, it requires 4.54 seconds on average to compute all component probabilities (and thus their entropy values) for the first configuration step in the Renault Megane example. In contrast, calling *satcount* repeatedly consumes 625.18 seconds.

Note that the first configurations steps are the most expensive in time. As the configuration process advances, the configuration space gets reduced and so the time needed to compute the probabilities. There is a point where both approaches converge and get response times close to 0.



(a) Renault Megane



(b) Electronic Shopping

Figure 5.3: Time required to compute component probabilities

[1]which is also the Achilles' heel for Chen et al.'s approach and Mazo et al.'s Heuristic 3.

CHAPTER
6
Conclusions

To satisfy a wide range of customers, product platforms must provide a high variety of optional components. For this reason, the configuration of all but trivial derivatives involves considerable effort in selecting which components they should include, while avoiding violations of the inter-component dependencies and incompatibilities. Our approach enriches existing automated configurators by reducing the number of steps required to configure a valid derivative.

Applying the Information Theory concept of entropy, our approach takes advantage of the fact that, due to the inter-component constraints, some decisions may be automatically derived from other decisions previously made. So the order in which decisions are made has a strong influence on the number of decisions required to complete a configuration. Moreover, our approach does not provide a static ordering that the customer is forced to follow. On the contrary, it suggests orderings dynamically, reacting to the customer decisions. In addition, we have proposed an algorithm that efficiently computes the variable probabilities of a Boolean formula, supporting this way not only our approach but also other methods proposed in related work.

The Renault Megane and Electronic Shopping configuration benchmarks have been used to test the applicability of our approach and its effectiveness. In particular, it has been shown that our approach needs less configuration steps than related work.

Bibliography

[ACF10] J-M. Astesana, L. Cosserat, and H. Fargier. Constraint-based vehicle configuration: A case study. In *22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 68–75, Oct 2010. 14

[AFM02] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps-application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002. 25

[BFL13] Christian Bessiere, H[U+FFFD]ne Fargier, and Christophe Lecoutre. Global inverse consistency for interactive constraint satisfaction. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2013. 25

[BG11] Ebrahim Bagheri and Dragan Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3):579–612, 2011. 25

[BHvM+09] Armin Biere, Marijn J.H. Heule, Hans van Maaren, Toby, and Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. 12, 13

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986. 12, 14

[CE11] Sheng Chen and M. Erwig. Optimizing the product derivation process. In *15th International Software Product Line Conference*, pages 35–44, Munich, Germany, 2011. IEEE Computer Society. 2, 9

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *6th International Workshop on Variability Modeling of Software-Intensive Systems*, pages 173–182, New York, NY, USA, 2012. ACM. 5

[CO08] Hadrien Cambazard and Barry O'Sullivan. Reformulating positive table constraints using functional dependencies. In *14th International Conference on Principles and Practice of Constraint Programming*, pages 418–432, Sydney, Australia, 2008. Springer. 25

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM. 13

[dec93] Reuse-driven software processes guidebook, version 02.00.03. Technical Report SPC-92019-CMC, Software Productivity Consortium Services Corporation, 1993. 5

[DGR11] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. The dopler meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011. 9

[Gan12] Graeme Keith Gange. *Combinatorial Reasoning for Sets, Graphs and Document Composition*. PhD thesis, Department of Computing and Information Systems. The University of Melbourne, 2012. 25

[HHOW05] Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In *20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference*, Pittsburgh, Pennsylvania, USA, 2005. AAAI Press / The MIT Press. 25

[HHRV11] Abel Hegedus, Akos Horvath, Istvan Rath, and Daniel Varro. A model-driven framework for guided design space exploration. In *26th International Conference on Automated Software Engineering*, pages 173–182, Washington, DC, USA, 2011. IEEE Computer Society. 9

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004. 2, 14

[HT07] Esben Rune Hansen and Peter Tiedemann. Compressing configuration data for memory limited devices. In *22nd National Conference on Artificial Intelligence*, Vancouver, British Columbia, Canada, 2007. AAAI Press. 25

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis. 2nd edition*. The MIT Press, 2012. 2

[Jan10] Mikolas Janota. *SAT Solving in Interactive Configuration*. PhD thesis, Department of Computer Science. University College Dublin, 2010. 9

[Jen04] R. M. Jensen. CLab: a C++ library for fast backtrack-free interactive product configuration. In *10th International Conference on Principles and Practice of Constraint Programming*, Toronto, Canada, 2004. Springer. 25

[Jun06] Ulrich Junker. *Handbook of Constraint Programming*, chapter Configuration, pages 837–868. Francesca Rossi and Peter van Beek and Toby Walsh, 2006. 9

[KCH$^+$90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. 5

[Kro12] Christian Kroer. Sat and smt-based interactive configuration for container vessel stowage planning. Master's thesis, IT University of Copenhagen, 2012. 25

[KZK10] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. Model counting in product configuration. In *1st International Workshop on Logics for Component Configuration*, pages 44–53, Edinburgh, UK., July 2010. 2, 13

[Lau06] Sean Quan Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, Dept. Electrical and Computer Engineering, University of Waterloo, Canada, 2006. 2, 25

[MDSD14] R. Mazo, C. Dumitrescu, C. Salinesi, and D. Diaz. *Recommendation Systems in Software Engineering*, chapter Recommendation Heuristics for Improving Product Line Configuration Processes. Springer-Verlag Berlin Heidelberg, 2014. 2, 9

[Men09] Marcilio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009. 9, 25

[NW07] 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, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc. 25, 26

[OOF05] Barry O'Sullivan, Barry O'Callaghan, and Eugene C. Freuder. Corrective explanation for interactive constraint satisfaction. In *19th International Joint Conference on Artificial Intelligence*, pages 1531–1532, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. 25

[PBL05] Klaus Pohl, Gunter Bockle, and Frank Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. 10

[PH04] F. K. Pil and M Holweg. Mitigating product variety's impact on the value chain. *Interfaces*, 34(5):394–403, 2004. 1

[POS$^+$12] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012. 25

[Que11] Matthieu Queva. *A Framework for Constraint-Programming based Configuration*. PhD thesis, Technical University of Denmark, 2011. 25

[SAH$^+$11] Samaneh Soltani, Mohsen Asadi, Marek Hatala, Dragan Gasevic, and Ebrahim Bagheri. Automated planning for feature model configuration based on stakeholders' business concerns. In *26th International Conference on Automated Software Engineering*, pages 536–539, Washington, DC, USA, 2011. IEEE Computer Society. 9

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948. 2, 11

[SHN$^+$07] Carsten Sinz, Albert Haag, Nina Narodytska, Toby Walsh, Esther Gelle, Mihaela Sabin, Ulrich Junker, Barry O'Sullivan, Rick Rabiser, Deepak Dhungana, Paul Grünbacher, Klaus Lehner, Christian Federspiel, and Daniel Naus. Configuration. *IEEE Intelligent Systems*, 22:78–90, 2007. 9, 12

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007. Feature Interaction. 5

[SRK$^+$11] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, pages 1–31, June 2011. 9

[SSJ05] Timothy W. Simpson, Zahed Siddique, and Jianxin Roger Jiao. *Product Platform and Product Family Design: Methods and Applications*. Springer, 2005. 1

[Ste80] Louis Steinberg. Question ordering in a mixed intiative program specification dialogue. In *1st Annual National Conference on Artificial Intelligence*, Stanford University, August 1980. AAAI Press. 9

[SW98] Daniel Sabin and Rainer Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems*, 13(4):42–49, July 1998. 1, 9, 12

[TBK09] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *31st International Conference on Software Engineering*, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society. 12

[vNBvOS06] Christof van Nimwegen, Daniel Burgos, Herre H. van Oostendorp, and Hermina Schijf. The paradox of the assisted user: guidance can be counterproductive. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 917–926, New York, USA, 2006. ACM. 2

[WBS$^+$10] J. White, D. Benavides, D.C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094–1107, 2010. 9

[WDSB09] Jules White, Brian Dougherty, Doulas C. Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *13th International Software Product Line Conference*, pages 11–20, Pittsburgh, USA, 2009. Carnegie Mellon University. 5

ACRONYMS

ADC	Automated Driving Controller
ANOVA	ANalysis Of VAriance
BDD	Binary Decision Diagrama
C	Car
CAB	Collision Avoidance Braking
EA	Enhanced Avoidance
FD	Features Diagram
FRF	Forward Range Finder
LRF	Lateral Range Finder
HSD	Honest Significant Differences
PP	Parallel Parking
S	Sensor
SA	Standard Avoidance