



**Jorge Miguel Sousa Barreiros**

M.Sc.

## **User-centric Product Derivation in Software Product Lines**

Dissertação para obtenção do Grau de Doutor em  
Informática

Orientadora : Professora Doutora Ana Maria Diniz Moreira,  
Professora Associada, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Dr. Nuno Manuel Robalo Correia

Arguentes: Prof. Dr. Alexander Egyed  
Prof. Dr. João Carlos Pascoal Faria

Vogais: Prof. Dr. João Miguel Fernandes  
Prof. Dr. António Rito Silva  
Prof. Dr. João Baptista da Silva Araújo Junior  
Prof.<sup>a</sup> Dra. Ana Maria Diniz Moreira



## **User-centric Product Derivation in Software Product Lines**

Copyright © Jorge Miguel Sousa Barreiros, Faculdade de Ciências e Tecnologia,  
Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.



*Para o Diogo, Joana e Nanda.*



# Agradecimentos

A obtenção do grau de Doutor é uma tarefa exigente, só tornada possível com a ajuda daqueles que nos são próximos. Por isso, é devido um agradecimento a todos os que me apoiaram nesta tarefa.

Não posso deixar de mencionar, em primeiro lugar, o Diogo e a Joana. Muito do tempo gasto neste trabalho foi-vos diretamente roubado. Por isso, ele é, em parte, tanto vosso como meu. Outra palavra especial vai para a Nanda, que foi a minha âncora ao longo destes anos. Estes agradecimentos especiais estendem-se a toda a minha família. Muito obrigado a todos!

Este trabalho também não teria sido possível sem a ajuda inestimável da minha orientadora, que alia, numa combinação rara e feliz, qualidades científicas, profissionais e humanas excecionais. Foi um verdadeiro privilégio trabalhar sob a sua supervisão. Muito obrigado pela sua disponibilidade, tempo, atenção e apoio incansável!

O meu apreço dirige-se também a todos aqueles que, de uma forma ou de outra, me ajudaram ao longo destes anos. São demasiados para enumerar, mas não fica esquecida a ajuda dos meus colegas de trabalho e do grupo de investigação, de todos aqueles que participaram nas minhas experiências de validação, e de todos os que contribuíram com observações, sugestões e críticas. Muito obrigado e um enorme abraço coletivo a todos!

Varias instituições contribuíram também de forma fundamental para o sucesso deste trabalho, tais como o Instituto Politécnico de Coimbra, o Instituto Superior de Engenharia de Coimbra, o Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, o Centro de Informática e Tecnologias da Informação e o NOVA LINCS. Não posso tampouco esquecer o apoio prestado pela Fundação para a Ciência e Tecnologia, consubstanciado

através das bolsas SFRH/BD/38808/2007 e SFRH/PROTEC/49834/2009. A todas estas instituições, e aos seus responsáveis, os meus sinceros agradecimentos.

Obrigado a todos!

*Jorge*

This work has been partially supported by the Portuguese Science and Technology Foundation, through grants SFRH/BD/38808/2007 and SFRH/PROTEC/49834/2009.



# Abstract

Software Product Line (SPL) engineering aims at achieving efficient development of software products in a specific domain. New products are obtained via a process which entails creating a new configuration specifying the desired product's features. This configuration must necessarily conform to a variability model, that describes the scope of the SPL, or else it is not viable. To ensure this, configuration tools are used that do not allow invalid configurations to be expressed.

A different concern, however, is making sure that a product addresses the stakeholders' needs as best as possible. The stakeholders may not be experts on the domain, so they may have unrealistic expectations. Also, the scope of the SPL is determined not only by the domain but also by limitations of the development platforms. It is therefore possible that the desired set of features goes beyond what is possible to currently create with the SPL. This means that configuration tools should provide support not only for creating valid products, but also for improving satisfaction of user concerns.

We address this goal by providing a user-centric configuration process that offers suggestions during the configuration process, based on the use of soft constraints, and identifying and explaining potential conflicts that may arise. Suggestions help mitigating stakeholder uncertainty and poor domain knowledge, by helping them address well known and desirable domain-related concerns. On the other hand, automated conflict identification and explanation helps the stakeholders to understand the trade-offs required for realizing their vision, allowing informed resolution of conflicts.

Additionally, we propose a prototype-based approach to configuration, that addresses the order-dependency issues by allowing the complete (or partial) specification of the features in a single step. A subsequent resolution process will then identify possible repairs, or trade-offs, that may be required for viabilization.

**Keywords:** *Software Product Lines, Feature Modeling, Product Derivation, Configuration Support, Soft Constraints*

# Resumo

O objectivo da engenharia de Linhas de Produtos de Software (do inglês *Software Product Lines*, ou SPL) tem como objetivo o desenvolvimento eficiente de produtos de software para um domínio específico. Produtos novos são obtidos através de um processo que obriga à criação de uma nova configuração, que descreve as características (em inglês, *features*) que devem ser integradas no produto. Esta configuração deve necessariamente estar de acordo com um modelo de variabilidade, que descreve o âmbito do SPL. Caso contrário, a configuração é inválida e não pode ser produzida. Sendo assim, para assegurar a viabilidade do produto, as ferramentas de configuração não permitem a especificação de configurações inválidas.

Uma questão distinta, no entanto, é assegurar que o produto que está a ser criado serve os interesses dos utilizadores (*stakeholders*, em inglês) tão bem quanto possível. Os utilizadores não são necessariamente peritos no domínio, pelo que podem ter expectativas irrealistas. Para além disso, o âmbito do SPL é determinado não só pelo domínio em questão, mas também pelas limitações das plataformas de desenvolvimento. É por isso possível que um cliente deseje um produto que inclua um conjunto inadmissível de características. Isto significa que, para além de se preocupar com questões de validade, as ferramentas de configuração deverão também oferecer apoio direcionado para promover a satisfação dos utilizadores. Este apoio deve ajudá-los a fazer os compromissos necessários, de forma a obter uma solução exequível que satisfaz as suas necessidades tão bem quanto possível.

Propomos atingir este objetivo através de um processo de configuração centrado no utilizador, que oferece sugestões de configuração, e também identifica e explica eventuais conflitos. As sugestões ajudam a mitigar a falta de conhecimentos ou incerteza do utilizador, ajudando-o a atingir uma solução com

propriedades desejáveis. Por outro lado, a identificação e explicação automatizada de conflitos ajuda o utilizador a melhor compreender os compromissos necessários para realizar a sua visão. Este apoio ao processo de configuração é conseguido através do uso de restrições suaves (em inglês, *soft constraints*), para representar informação de variabilidade e restrições do utilizador.

Propomos também uma abordagem de configuração baseada em prototipagem, que ajuda a ultrapassar a dependência do resultado final em relação à ordem de configuração das características. Esta abordagem permite realizar a especificação completa (or parcial) da configuração de várias características num único passo. Isto pode resultar numa configuração inválida, pelo que um processo subsequente irá identificar ações de reparação possíveis, que viabilizam da configuração pretendida.

**Palavras-chave:** *Linhas de Produtos de Software, Modelação de Características, Derivação de Produtos, Suporte de Configuração, Restrições Suaves*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Objectives and Challenges . . . . .	2
1.2	Research Methodology . . . . .	6
1.3	Proposed Solution . . . . .	7
1.3.1	Configuration Advisor and Soft Constraints . . . . .	7
1.3.2	Prototype-based Configuration Approach . . . . .	9
1.3.3	Earlier Work . . . . .	9
1.4	Evaluation . . . . .	10
1.5	Contributions . . . . .	11
1.6	Structure of the Document . . . . .	11
<b>2</b>	<b>Feature Modeling and Product Derivation</b>	<b>13</b>
2.1	Software Product Lines . . . . .	13
2.2	Feature Models . . . . .	15
2.2.1	Boolean Logic Representation of Feature Models . . . . .	18
2.2.2	Feature Model Configuration . . . . .	19
2.3	Iterative Configuration of Feature Models . . . . .	20
2.4	Assisting the Decision Process . . . . .	23
<b>3</b>	<b>Boolean Soft Constraints</b>	<b>25</b>
3.1	Soft Constraints in Feature Modeling . . . . .	25
3.2	Boolean Soft Constraints in Feature Models . . . . .	26
3.3	Normative Semantics . . . . .	28
3.3.1	On Impossibility Functions . . . . .	31
3.3.2	Annotation of a Feature Model with Normative Soft Constraints . . . . .	34

3.4	Annotational Semantics . . . . .	34
3.5	Conclusions . . . . .	35
<b>4</b>	<b>Soft Constraints in Domain Engineering</b>	<b>37</b>
4.1	Domain-related Soft Constraints . . . . .	37
4.2	Prototypical Applications . . . . .	40
4.2.1	Soft Constraint Annotation Patterns . . . . .	40
4.2.2	Optional Selection Suggestion . . . . .	40
4.2.3	Reversed Constraint Suggestion . . . . .	41
4.2.4	Group Selection Suggestion . . . . .	42
4.2.5	Soft Constraints and Feature Model Evolution . . . . .	43
4.3	Suspicious Soft Constraint Interactions . . . . .	45
4.3.1	Suspicious Interaction Classification . . . . .	46
4.3.2	Identification of Suspicious Interactions . . . . .	48
4.4	Conclusions . . . . .	52
<b>5</b>	<b>Enhanced Configuration Support</b>	<b>55</b>
5.1	Enhanced Support Overview . . . . .	55
5.1.1	Configuration Suggestions . . . . .	56
5.1.2	Conflict Identification and Explanation . . . . .	57
5.2	Algorithms for Configuration Advice and Conflict Analysis . . . . .	60
5.3	Tool Description . . . . .	63
5.4	Conclusions . . . . .	66
<b>6</b>	<b>Prototype-Based Configuration</b>	<b>69</b>
6.1	Prototype-based vs. Iterative Configuration . . . . .	70
6.2	Configuration Repair Overview . . . . .	75
6.3	Configuration Repair Based on Cover Information . . . . .	76
6.3.1	Cover and Literal Minimization . . . . .	77
6.3.2	Feature Model Partitioning for Efficient Cover Computation . . . . .	77
6.3.3	Configuration Repair Using Cover Information . . . . .	80
6.3.4	Performance and Optimality . . . . .	83
6.3.5	Selection Criteria . . . . .	84
6.3.6	Repair of Partitioned Feature Models . . . . .	84
6.4	Presentation of Potential Repairs . . . . .	87
6.5	Tool Description . . . . .	88
6.6	Conclusions . . . . .	90

<b>7</b>	<b>Validation</b>	<b>93</b>
7.1	Identification of Suspicious Interactions . . . . .	94
7.1.1	Experiment Objectives and Goals . . . . .	94
7.1.2	Data Set Construction and Constraint Injection . . . . .	95
7.1.3	Unsatisfiable and Untriggerable Soft Constraint Identification Experiment . . . . .	99
7.1.4	Contradictory Soft Constraint Identification Experiment . . . . .	100
7.2	Configuration Repair Testing . . . . .	103
7.2.1	Experiment Objectives and Goals . . . . .	103
7.2.2	Partitioning and Cover computation . . . . .	105
7.2.3	Repairing Random Invalid Configurations . . . . .	109
7.2.4	Problem Decomposition . . . . .	114
7.3	Empirical Testing of Enhanced Configuration Support . . . . .	115
7.3.1	Experiment Design . . . . .	115
7.3.2	Data Analysis and Hypothesis . . . . .	117
7.3.3	Experiment Realization . . . . .	119
7.3.4	Statistical Analysis of Results . . . . .	119
7.3.5	User Feedback . . . . .	121
7.4	Results Discussion . . . . .	124
7.4.1	Identification of Suspicious Interactions . . . . .	124
7.4.2	Configuration Repair Testing . . . . .	125
7.4.3	Empirical Testing of Enhanced Configuration Support . . . . .	127
7.5	Threats to Validity . . . . .	128
<b>8</b>	<b>Conclusions and Future Work</b>	<b>131</b>
8.1	Research Questions Revisited . . . . .	131
8.1.1	How to leverage the use soft constraints in SPL development to achieve this goal? . . . . .	132
8.1.2	How to provide enhanced configuration support? . . . . .	133
8.1.3	How to represent the user's idealized configuration? . . . . .	135
8.1.4	How effective is enhanced configuration support? . . . . .	137
8.1.5	How efficient is enhanced configuration support? . . . . .	137
8.2	Past, Present, and Future . . . . .	138
8.2.1	The Past . . . . .	138
8.2.2	The Present . . . . .	140
8.2.3	The Future . . . . .	141
	<b>Appendices</b>	<b>155</b>

<b>A</b>	<b>Earlier Work</b>	<b>157</b>
A.1	Graphical Representation of Configuration Knowledge . . . . .	157
A.2	Reusable Model Slices . . . . .	162
A.2.1	Overview . . . . .	162
A.2.2	Wildcards . . . . .	163
A.2.3	Default values . . . . .	163
A.2.4	Operations . . . . .	163
A.2.5	Instantiation . . . . .	164
<b>B</b>	<b>Box and Whiskers Plots</b>	<b>167</b>
B.1	Box and Whiskers Plots . . . . .	167
<b>C</b>	<b>Validation Results</b>	<b>171</b>
<b>D</b>	<b>GQM Document</b>	<b>179</b>
<b>E</b>	<b>Experiment Test Cases</b>	<b>183</b>

# List of Figures

1.1	From stakeholder wishes to an implementable configuration . . . . .	3
1.2	Configuration advisor . . . . .	8
2.1	Feature tree elements . . . . .	17
2.2	Mobile phone feature model . . . . .	18
2.3	Example of BDD and its use for configuration purposes. . . . .	21
2.4	Feature model for configuration example . . . . .	21
3.1	Normative soft constraints example (the key describes the condensed variable notation used in the text) . . . . .	30
3.2	Complex Normative Constraint Example . . . . .	32
4.1	Feature model describing vehicles configuration . . . . .	39
4.2	Example of Optional Selection Suggestion . . . . .	41
4.3	Example of Reverse Constraint Suggestion . . . . .	42
4.4	Example of Group Selection Suggestion . . . . .	43
4.5	Evolving a feature model via soft constraints . . . . .	44
4.6	Restructuring the feature tree via soft constraints . . . . .	45
4.7	Unsatisfiable soft constraint example . . . . .	46
4.8	Contradictory soft constraints . . . . .	46
4.9	Untriggerable soft constraint Example . . . . .	47
5.1	Enhanced configuration support example . . . . .	58
5.2	Features conflicted due to multiple constraints . . . . .	59
5.3	Configurator tool packages . . . . .	64
5.4	Configurator tool - Experiment test case . . . . .	65
6.1	Feature model and idealized configuration . . . . .	71

6.2	Order-dependent outcome in iterative configuration . . . . .	72
6.3	Prototype configuration . . . . .	72
6.4	Configuration repair example . . . . .	75
6.5	Feature model decomposition example . . . . .	78
6.6	Feature model decomposition example - Step 1 . . . . .	78
6.7	Feature model decomposition example - Step 2 . . . . .	79
6.8	Feature model decomposition example - Step 3 . . . . .	80
6.9	Feature model for example of repair extraction from cover terms . .	81
6.10	Invalid configuration . . . . .	82
6.11	Configuration repair tool in action . . . . .	89
7.1	Analysis time per constraint . . . . .	101
7.2	Analysis time per feature model . . . . .	102
7.3	Percentage of contradictory pairs . . . . .	104
7.4	Analysis time for all contradictions with one specific constraint [ms]	105
7.5	Analysis time of a pair of constraints [ms] . . . . .	106
7.6	Ratio between the number of terms after partitioning and before partitioning ( $T_P/T_F$ ) (outliers not represented) . . . . .	108
7.7	Time required for execution of partitioning algorithm . . . . .	110
7.8	Time to find first repair (ms). Outliers not represented. . . . .	112
7.9	Academic and industrial experience of participants (some partici- pants are not represented as they chose not to provide the corre- sponding optional information). . . . .	120
7.10	Self-assessment of participant competence . . . . .	120
7.11	Feedback results, describing participant perception of efficiency and effectiveness gains, as well as helpfulness of trade-off infor- mation. . . . .	122
A.1	Media application example . . . . .	157
A.2	Configuration module . . . . .	158
A.3	Association describing configuration flow and cardinality constraints	159
A.4	Composition example . . . . .	159
A.5	Specialization example . . . . .	160
A.6	Association describing constraint only, without any configuration flow. . . . .	160
A.7	n-Ary association to represent complex constraint . . . . .	161
A.8	Representing alternative implementations with specializations. . .	161
A.9	Example of aspectual model slice . . . . .	162

A.10 Resulting MATA transformations . . . . .	166
B.1 Box and whiskers plot example . . . . .	168



# List of Tables

2.1	Feature model transformation into Boolean propositional logic . . .	18
2.2	Feature model transformation example . . . . .	19
3.1	Boolean soft constraints . . . . .	27
3.2	Normative constraint example results . . . . .	30
7.1	Feature models included in input data set. . . . .	96
7.2	Injection results . . . . .	99
7.3	Aggregated results for unsatisfiable and untriggerable soft constraint identification . . . . .	100
7.4	Extract of the results for largest models . . . . .	107
7.5	Partitioning - Outlier cases . . . . .	109
7.6	Repair of random configurations — excerpt of full results . . . . .	111
7.7	Extract from repair results — Finding repairs for invalid configurations that preserve selected features . . . . .	113
7.8	Problem decomposition of repairs of random configurations. . . . .	114
7.9	Feature models selected for test cases . . . . .	117
7.10	Statistics for self-assessment of participant competence . . . . .	121
7.11	Aggregated feedback results . . . . .	122
C.1	Results for soft constraint injection . . . . .	172
C.2	Results for identification of untriggerable and unsatisfiable soft constraints . . . . .	173
C.3	Results for identification of contradictory pairs of soft constraints .	174
C.4	Partitioning and cover extraction results . . . . .	175
C.5	Repair results - Finding repairs that minimize Hamming distance from random invalid configurations. . . . .	176

C.6 Repair results - Finding repairs for invalid configurations preserv- ing selected features. . . . .	177
--	-----

# List of Algorithms

1	Suggestion computation . . . . .	60
2	Active feature computation . . . . .	61
3	Analyze inconsistency . . . . .	62
4	Partition feature tree . . . . .	79
5	Computing a candidate repair from a specific term . . . . .	82
6	Computing all candidate repairs . . . . .	83
7	Soft constraint injection algorithm . . . . .	98



# Glossary

**Application Engineering:** In **software product line** engineering [Cle01, PBL05], it is the process by which a customized product is developed by creating a valid **configuration** of **features**, composing reusable core assets and developing whatever application-specific assets may be required. This process is conducted by the Application Engineer.

**Configuration:** In **software product line** engineering [Cle01, PBL05], it is a set of selected and deselected **features** [KKL<sup>+</sup>98]. If a configuration conforms to the **feature model**, it is said to be *valid*, otherwise it is *invalid*. A *partial* configuration is one not yet complete, hence including open **features** (that are not yet selected or deselected). A valid configuration corresponds to one of the specific products that may be created by the **software product line**.

**Constraint:** A **feature model** may include constraints, represented either graphically or textually [KKL<sup>+</sup>98, Bat05]. Typically, these are *hard* constraints, that must be upheld in all valid products. Alternatively, some constraints may be *soft* [BM11, BDNRG10]. Soft constraints represent preferential **configuration** options and, unlike hard constraints, do not necessarily impair validity of a **configuration** if not satisfied.

**Domain Engineering:** In **software product line** engineering [Cle01, PBL05], domain engineering is the process by which the development platform is developed and maintained. The domain engineer defines the scope of the **software product line** by defining the variability model. He is also responsible for creating and maintaining the reusable core assets.

**Feature:** Features are composable artefacts that modularize modifications of underlying structures (program, architecture, requirements or other software development artefacts) satisfying some stakeholder's needs. They are usually aligned with a user-visible aspect or characteristic of the domain [KKL<sup>+</sup>98, CHE05, Cle01, PBL05, Bat06].

**Feature Model:** Feature models, introduced in [KKL<sup>+</sup>98], identify valid product **configurations** by using an hierarchic **feature** tree describing optional and mandatory **features**, as well as **feature** groups. They can be annotated with additional **constraints**, represented graphically (e.g., linking dependent **features** with a dependency arrow) or by textual annotations. Feature models are frequently used in **software product line** engineering for identifying valid **configurations** corresponding to feasible products, or variants [KLD02, Cle01].

**Software Product Line:** Software product line engineering can be defined as a paradigm to develop software applications using platforms and mass customization [Cle01, PBL05]. A platform is, in this context, a software infrastructure that allows new software products to be efficiently developed and produced. Mass customization entails efficient tailoring and development of the software products according to individualized client needs.

**User:** In this work, user refers to the role of whoever creates a **configuration**, using appropriate tool support. Typically, such responsibility lies with the **application engineer** on behalf of the stakeholder, however, we also envision the possibility of direct stakeholder action.



# Introduction

Software users' needs and desires change continuously. Unless specific provisions are made, keeping up with these evolving demands can entail considerable development efforts. Addressing evolution in a timely way with reasonable effort is a major goal in Software Engineering. To achieve this, Software Product Line (SPL) engineering has been proposed [Cle01, PBL05, KLD02]. SPLs leverage planned reuse of development artifacts to allow for the timely development of alternative software products in a specific domain, in response to evolving users' wishes.

Customized new products are created by a configuration process, which entails selecting the features to include and then composing the associated artefacts. Typically, not all combinations of features are viable: a description of commonality and variability, usually in the form of a feature model [KCH<sup>+</sup>90], describes valid configuration choices. The configuration process can be supported by automated tools providing services such as configuration completion and choice propagation. These techniques ensure feasibility by allowing only valid products to be created.

However, little support is provided for helping the developer creating a product best meeting stakeholders' needs. This is not a trivial task due to potential misalignments between stakeholders' desires and the capabilities of the SPL, manifested as restrictions on the admissible features that can integrate a product. Also, conflicting stakeholder wishes may exist. For example, if the intended configuration is specified by multiple stakeholders, conflicting specifications may be

created. Alternatively, even a single stakeholder may specify an invalid configuration if he does not have a clear comprehension of the domain or the limitations of the SPL. These conflicts need to be resolved somehow before an actual product can be created.

The issue is, then, finding ways to facilitate the creation of products that are the best possible match to the stakeholders' desires. This is a complex problem that can be approached from different perspectives, such as SPL evolution (how to evolve the SPL to better accommodate user wishes), requirement engineering (how to elicit and model volatile and evolving user requirements), architecture definition (how to create a flexible architecture that efficiently supports all the required variability and allows evolution), product preview and visualization techniques (how to provide feedback to the user so he has a clearer understanding of the product being created). Addressing such a diverse range of issues within the context of a single work is not feasible. For the sake of focusing our efforts, our approach is then concerned with providing mechanisms that assist the stakeholder (or a domain engineer on his behalf) during the product configuration process, attempting to ensure that the desired and actual configuration are as close as possible. While ideally these configurations would be identical, this is not always the case. The former can be perceived as a manifestation of the stakeholder's desires, while the latter is a representation of a feasible product. In this way, dissimilarities correspond to required trade-offs or compromises.

The remainder of this chapter is structured as follows. Section 1.1 presents the objectives and challenges faced by our work. Section 1.2 discusses our research methodology, while Section 1.3 provides an overview of our proposed solution. Section 1.4 concerns our approach to validate our results, and our contributions are enumerated in Section 1.5. Section 1.6 presents the structure of this thesis.

## 1.1 Objectives and Challenges

Although SPLs reduce the time required to create variant products, an important question is to determine the features of the product that best match the stakeholder's intention within the constraints of the variability model. Feature models have originally been proposed to represent a decomposition of system functionalities from a user's perspective [KCH<sup>+</sup>90]. Within this paradigm, users' desires can then be naturally represented as a specific configuration of features that satisfies their needs and expectations. However, preferences and desires are dependent on users' volition and do not necessarily conform to the constraints of the

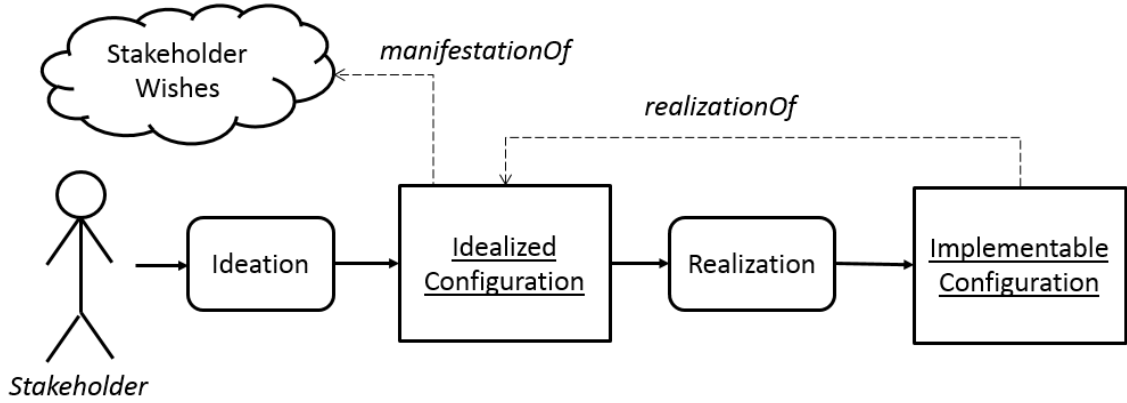


Figure 1.1: From stakeholder wishes to an implementable configuration

variability model of the the SPL. If the intended configuration is interpreted as an additional constraint applied to the feature model, then it may be the case that the product specification becomes overconstrained, making the users' intended configuration unfeasible. So, in the general case, it cannot be ensured that all features may be selected according to the preferred users' choices.

Figure 1.1 presents our model of the process that culminates with the creation of a viable configuration corresponding to stakeholders' wishes. As illustrated, an implementable configuration is obtained only after a process where the stakeholders' wishes are first codified into an *idealized* configuration<sup>1</sup> (in a process we designate as *ideation*), which is then subsequently transformed into an *implementable* configuration (that is, one that is viable, according to the variability model) by the *realization* process<sup>2</sup>.

Stakeholders do not necessarily have all the domain knowledge required to understand all the complexities and intricacies that impact on the feasibility of their desired product. This means that the idealized configuration may be unfeasible. For example, the stakeholder may wish a product that includes incompatible features. These features may be incompatible due to technicalities that are well outside his field of expertise. Alternatively, it may be a business-level decision from the SPL owner not to provide certain combinations of features, regardless of technical viability.

Conversely, stakeholders may also be unaware of some viable functionalities

<sup>1</sup>Whether or not such an artifact actually materializes or exists only as a mental construction in the stakeholders' (or application engineers') mind is irrelevant to our discussion. Most frequently, SPL literature does not acknowledge this distinction and assumes that the implementable configuration corresponds to a direct manifestation of stakeholders' intentions.

<sup>2</sup>Although for simplicity sake we represent ideation and realization as sequential processes whose outputs are complete configurations, it is also possible to consider the same processes being applied iteratively over the time to partial configurations incrementally growing to completeness.

that would be of their interest. These would not be included in the idealized configuration due to obliviousness, and therefore it would be sub-optimal. An example of a sub-optimality would be one where the stakeholder creates a configuration that does not fully exploit the potential of the system in ways that would be beneficial to him due to lack of domain expertise, like not including security enhancing features in a distributed multi-user system. Other example of sub-optimal idealization may happen when the stakeholder attempts to create a feasible idealized configuration, but has outdated knowledge of the SPL capabilities. He may, for instance, be unaware of new features that have been recently introduced, or of constraints that have been lifted. In this case, idealization may be polluted by ungrounded concerns of feasibility, as the stakeholder will abstain from using new features, or voluntarily force himself to respect obsolete constraints that no longer actually apply.

This means that any idealized configuration may be both unfeasible and sub-optimal. That is, the output of the ideation process (the idealized configuration) may not accurately reflect the stakeholder's wishes, and it may also be invalid. If the idealized configuration is not valid, the ensuing realization process will transform it into a configuration that is actually implementable. The realization process involves sanitizing inconsistencies in the idealized configuration to obtain a new valid configuration, that is as "close" as possible to the idealized configuration.

The intermediation of an application engineer (the role of the developer in charge of creating a specific product, see Section 2.1), does not make these difficulties disappear: although the application engineer is most likely fairly conversant with the variability model and the possibilities and boundaries of the SPL, resolution of incompatibilities is a challenging problem that ultimately requires stakeholder intervention. Taking advantage of untapped possibilities is also problematic, as the application engineer would have to discern if some of the unsolicited functionalities would actually be of use or interesting for the stakeholder (again stakeholder intervention would still be ultimately required). In this way, for convenience we henceforth refer only to the stakeholder (or simply "user" for convenience) as having an active role during the configuration process, with the understanding that the application engineer might also play a more or less significant part in the process.

While the stakeholder may have a deep knowledge of the domain of interest and the capabilities of the SPL, this is hardly the case in all scenarios. Since the problems we are addressing are exacerbated by the lack of domain knowledge, it

is important that our approach does not presume that the stakeholder is a domain expert. We must therefore seek ways to overcome these challenges without overly relying on stakeholder's expertise.

These problems may be summarized in two points:

- The idealized configuration may be sub-optimal. This means that it does not reflect the stakeholders wishes as accurately as possible.
- The idealized configuration may be unfeasible. It is necessary to deviate from the idealized configurations, and trade-off decisions are most likely necessary.

Sub-optimality is related to deviation from preferential configuration options, while unfeasibility can be traced to overconstrainment. *Soft constraints* are a prime candidate for representing both preferential and overconstrained configurations, and will therefore play a significant role in our solutions.

Assuming a relatively uninformed stakeholder is in question, these issues can be synthesized into the following main research question

**How to support the derivation of software products  
in SPL that best conform to stakeholders' goals?**

which can be sub-divided into the following questions

1. **How to leverage the use soft constraints in SPL development to achieve this goal?**
  - (a) How to use soft constraints in the domain engineering step?
    - i. How can suspicious soft constraint interactions be efficiently identified and reported to the domain engineer?
    - ii. How common are these interactions?
  - (b) How to use soft constraints in the application engineering step?
2. **How to provide enhanced configuration support?**
3. **How to allow users to model, and inform the system, their idealized configuration, so that automated support can be given?**
  - (a) How to handle possibly overconstrained and invalid idealizations?

- (b) How to properly address the large number of potential trade-offs that may be possible, without overwhelming the user with a very large number of alternative possibilities?

4. **How effective is enhanced configuration support?**

5. **How efficient is enhanced configuration support?**

These research questions are revisited and discussed in the light of the content of the next chapters in Section 8.1.

## 1.2 Research Methodology

Our approach was based on the Technology Research method [SS07]. This method entails the production of new or improved technological artefacts to address a specific need. These artefacts are then demonstrated to be suitable for their purpose. The main steps of this process are:

1. **Problem Analysis.** Where the researcher identifies the potential need for new or improved technological artefacts. Based on the literature review and the analysis of existing approaches for SPL, we identified the problems and challenges discussed in Section 1.1.
2. **Innovation.** Where the researcher explains how to make an artifact that satisfies the needs identified in the problem analysis step, and creates it. After having identified the main problem areas, we began identifying some possible ways of addressing them. This resulted in the publication of some early work (Section 1.3.3). Feedback from review and presentation was used to further develop our approach, finally resulting in the solutions presented in Sections 1.3.1 and 1.3.2 and described throughout chapters 3 to 6. Prototype tools were created to support the validation efforts.
3. **Validation.** Where the researcher demonstrates that the artifacts created in the previous step address the needs. We conducted a validation process, involving prototype tools, data from public repositories, automated test case generation and empirical testing, described in Chapter 7.

## 1.3 Proposed Solution

Our solution addresses the research questions highlighted in Section 1.1. The first sub-question (question 1) is concerned with helping the user to derive the maximum benefit from the SPL capabilities by ensuring his idealized configuration is not sub-optimal, that is, it does not miss out on potentially interesting features and configuration possibilities he might not be aware of. The next sub-questions (questions 2 and 3) are concerned with ensuring that the realization process facilitates the creation of an implementable configuration that is close to the idealized, while ensuring the stakeholder understands the trade-offs required for validity sake and makes informed decisions. Research questions 4 and 5 make it clear that the proposed solution is expected to provide efficiency and effectiveness gains.

Our approach addresses these problems during the *realization* step (cf. Fig 1.1). Although the idealized configuration is already decided at this point (or at least part of it is: at minimum, the stakeholder must have decided to select or deselect just a single feature), during the realization phase it is possible to provide feedback that allows the user to retroactively change and adapt the idealized configuration if necessary. We present two alternatives, suited for different configuration approaches. One of these alternatives is based on a configuration advisor and soft constraints, represented in Figure 1.2, while the other is a prototype-based configuration approach.

Earlier work developed by the authors proposed a graphical representation of configuration knowledge [BM09b], supported by an associated aspect-oriented modelling approach [BM09c].

### 1.3.1 Configuration Advisor and Soft Constraints

The configuration advisor works in tandem with an incremental or staged configurator [CHE04], which allows the user to specify the implementable configuration incrementally, propagating these choices as required to ensure validity is preserved. During each step, the advisor may provide feedback to the stakeholder. He may then disregard this advice or act upon it and update the idealized configuration accordingly. This feedback is provided within the context of the feature model and is based on the partial implementable configuration, domain information and a representation of the idealized configuration. The latter two are optional, but at least one must be available so that advice can actually be generated.

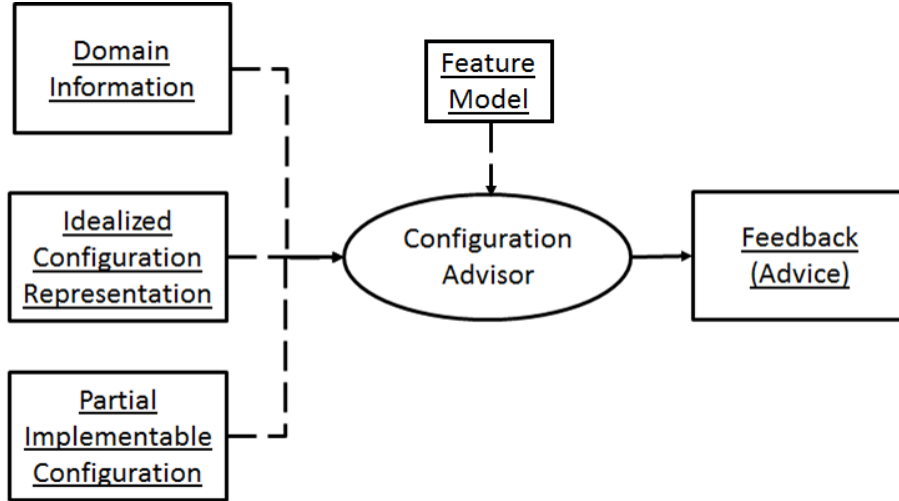


Figure 1.2: Configuration advisor

In a staged configuration approach, the implemented configuration will incrementally grow step-by-step. At each one of these steps, advice may be generated that is dependent on earlier choices made by the user. This advice is concerned only with the unspecified portion of the implemented configuration. That is, the advisor may suggest that some yet open (unspecified) feature should be selected. However, it will never advise changing the selection status of a feature that is already (de)selected.

Domain information, if available, is used to provide configuration suggestions that may cause the user to change his original intentions, that is, his intended configuration, to better take advantage of SPL capabilities. Referring to the discussion in Section 1.1, it is used to avoid or reduce sub-optimality. If an explicit representation of the idealized configuration exists, it may be provided to the advisor. The main purpose of this will be to use advisor feedback to identify and guide the stakeholder through potential trade-offs that must be made to realize an invalid idealized configuration.

One important question is determining what is the representation of idealized configuration and relevant domain information. The idealized selection state of each feature can then be represented by a *soft constraint* applied to the feature model suggesting, but not requiring, selection (or deselection) of the feature [BM12]. Soft constraints represent desired or common properties that should, but must not, be held by the configured product. Straightforward application of this principle will result in a set of soft constraints, one for each (de)selected feature in the idealized configuration. Soft constraints are more suited for representing the idealized configuration than hard constraints, since the necessary realization

process may prevent satisfaction of idealized preferences.

As a modeling tool, soft constraints can also be used to represent domain specific information. For example, a soft constraint may be used to indicate that two features are commonly jointly selected, or that at least one of a certain set of features is usually selected. Using a similar modeling approach for both user preferences and domain information allows an unified reasoning approach to address all cases.

### 1.3.2 Prototype-based Configuration Approach

While the advice-based approach is suitable for a staged configuration environment, our prototype-based approach follows a different strategy. In this case, a representation of a complete or partial idealized configuration (the prototype) must be provided by the stakeholder. The system will then proceed to automatically identify and list what changes and trade-offs are necessary to ensure the prototype becomes viable (thereby transforming it into an implementable configuration). These changes are presented to the stakeholder, who then has the responsibility of selecting his preferred resolution among all possible alternatives. Like the advise-based strategy described in the previous section, prototype-based configuration ensures that the stakeholder is guided during the realization process. By finding and presenting to the stakeholder all possible alternative ways in which the prototype may be changed into a viable configuration, in accordance to an optimization criteria (usually the smallest distance from prototype to its realization in terms of Hamming distance [Ham50]), the developer is provided with a complete and fairly exhaustive list of alternative changes. By selecting and applying one of those alternatives, the stakeholder is implicitly resolving the trade-off as stipulated by the option he choses.

At the core of this approach, we find our cover-based configuration repair algorithm [BM14a], which is able to efficiently compute a large number of alternative optimal repairs and present a condensed list of partitioned repair alternatives of manageable size to the user.

### 1.3.3 Earlier Work

Earlier work proposed to address the complexity of the configuration process by using a graphical method for describing configuration parameters dependencies and flow [BM09b]. This approach is centered on the use of configuration modules, which describe parameterizable implementation artefacts such as aspectual

model components or source code files. This model describes how configuration information impacts the selection and configuration of features and their implementation artefacts. It provides the graphical tools akin to a template-based approach to configuration and implementation [CE00] by representing associations and specializations. However, it is not specific to any programming language or implementation technique. Additional capabilities are also provided such as the representation of complex configuration dependencies via n-ary associations or support for specifying the enumeration of parameter lists. The capabilities of this representation technique can be observed in section A.1 of the appendices, where an example of one such model representing the configuration of a multimedia product is presented and described.

Another early work we developed proposes an aspect-oriented modeling technique [BM09c]. It was designed to complement and support the work in [BM09b], by offering generic configurable aspects that can be instantiated, using a simple language, into concrete MATA aspects [WJE<sup>+</sup>09], capable of advising correctly diverse base models. See appendix A.2 for an overview of the proposed technique.

## 1.4 Evaluation

We conduct a series of experiments to assess our work. Two types of experiments are used: runtime testing of our algorithms and empirical testing with users. The former allows us to identify and analyse the performance of our algorithms, while the latter provides real user feedback concerning the perceived advantages of our proposed techniques. All these evaluation efforts are based on the use of real (non-synthetic) feature models retrieved from a public repository [MBC09]. This lends a level of representativity and independence that is most welcome. However, on accordance to our proposal, these models must be extended to include some additional information. For the most part, rather than relying on manual annotation, we decided to resort to automated annotation, following certain guidelines to ensure a minimum level of plausibleness is maintained. Runtime testing of our techniques is based on these annotated models. These tests allow us to observe the performance profile of the algorithms and determine whether or not they are suitable for tool integration. We also use these tests to identify the prevalence of certain potential anomalies that are detected and identified by our algorithms. Empirical testing with users replicates a process of creating a configuration based on specific user preferences, and allows

actual human feedback to be considered for assessing the potential advantages of our work, in terms of efficiency and effectiveness of the configuration process. To conduct this experiment, we first identified the main objectives using a GQM document. Then, test cases were created and our prototype tool adapted to automatically submit test results. An online form was also made available for receiving user feedback. After a few adjustments based on an initial trial run feedback, several experiment sessions were conducted in two different locations. The experiment results were then subjected to statistical analysis.

## 1.5 Contributions

The main contributions of this work are:

1. Proposing and discussing the use of soft constraints throughout the SPL lifecycle, from domain analysis to product configuration [BM11, BM12] (Chapters 3 and 4).
2. Identifying and describing certain types of potentially anomalous soft constraint interactions, specifically unsatisfiable, untriggerable and contradictory soft constraints [BM12] (Chapter 4).
3. Providing mechanisms for automatically identifying and reporting these potentially anomalous interactions [BM12] (Chapter 4).
4. Proposing an algorithm that leverages soft constraints to offer enhanced support during the configuration step, allowing increased satisfaction rates and better comprehension of required trade-offs [BM13, BM14b] (Chapter 5).
5. Devising a novel configuration repair mechanism, with very high performance and especially well suited for repairing multiple invalid configurations for the same feature model [BM14a] (Chapter 6).

## 1.6 Structure of the Document

The remainder of this document is structured as follows:

- Chapter 2, **Feature Modeling and Product Derivation**. In this chapter, we present fundamental concepts of Software Product Lines, feature modeling and product derivation, that are relevant for the discussion of ensuing chapters.

- Chapter 3, **Soft Constraints in Feature Modeling**. In this chapter, we define our soft constraint model. Two different semantic categories of soft constraints are identified, and we discuss the suitability of each such category for our purposes.
- Chapter 4, **Soft Constraints in Domain Modeling**. In this chapter, we identify prototypical applications of soft constraints in the domain engineering step. These involve primarily the representation of domain information, but are also of use in feature model evolution.
- Chapter 5, **Enhanced Configuration Support**. In this chapter, we present our enhanced configuration support techniques based in soft constraints. We describe our advice generation and conflict detection algorithms, which are integrated into an incremental configurator to provide an enhanced configuration experience, and present our prototype configurator tool.
- Chapter 6, **Prototype-based Configuration**. In this chapter, we present a generalization of iterative configuration that allows the stakeholder to configure more than one feature simultaneously. Each such configuration step is then followed by an additional step where possible inconsistencies are detected and repaired.
- Chapter 7, **Validation**. In this chapter, we present the validation results. Results are based in publicly available feature models from online repositories. We have conducted tests to assess the runtime performance of our algorithms, and an empirical test was also conducted.
- Chapter 8, **Conclusions**. In this chapter, we conclude our work by revisiting and discussing the research questions in the light of the information garnered in earlier chapters. We also offer an overview of the current status of the work, present its evolution, and provide suggestions for future work.



# Feature Modeling and Product Derivation

Since their introduction over 20 years ago [KCH<sup>+</sup>90], feature models have been recurrently used to represent variability in Software Product Line engineering [Cle01, PBL05, WL99, Par76]. Therefore, to clarify the context of our work and establish technical bases, we present an overview of SPL engineering (Section 2.1). Several variant feature models are also described, as well as the formalization and graphical representation of Boolean feature models (Section 2.2). Product derivation entails creating a configuration that conforms to the feature model (Section 2.3). This configuration determines the properties of the product, and as such is a manifestation of the stakeholder's goals. Some techniques have been described to support the stakeholder in identifying the configuration that best satisfies those goals (Section 2.4).

No original contributions can be found in this chapter, as its purpose is establishing the required background for understanding later chapters.

## 2.1 Software Product Lines

One century ago, the adoption of assembly lines and pipelined manufacturing in the automotive industry heralded a new age in industrial manufacturing. It was a significant paradigm shift that eschewed earlier approaches in the search of

higher labor efficiency and productivity. Dramatic improvements were immediately observed, such as a nine-fold decrease in the time required to fully assemble a vehicle (from 12h30m down to 1h30m) [Hou85]. Since assembly lines excel at efficient production of high volumes of identical parts, product conceptualization and design also evolved accordingly. Improved tooling and production processes allowed finer precision and the creation of interconnectable, normalized and interchangeable parts. Consequently, rather than relying on custom-made components tailored specifically for a single specific product, the same parts were now being applied in the construction of a large number of different products [Hou85]. As a consequence, design and production efforts began focusing on product families or lines rather than individual products.

The significant benefits obtained from the application of product line development in industrial manufacturing inspired the search for similar solutions in other domains. Software Product Line (SPL) engineering attempts to leverage the same principles to achieve similar benefits in software development. According to Pohl [PBL05], it can be defined as a paradigm to develop software applications using platforms and mass customization. A platform is, in this context, a software infrastructure that allows new software products to be efficiently developed and produced. Mass customization entails efficient tailoring and development of the software products according to individual client needs. To address both aspects, SPL engineering branches into two interdependent but separated processes. The first one, known as *Domain Engineering*, is responsible for establishing and maintaining the platforms, while the second process, *Application Engineering*, concerns derivation of the actual products, using the platforms, according to customized requirements of clients.

In the Domain Engineering step, many types of reusable software artifacts can be produced (code, models, documentation, tests, etc.). These assets are created with the specific purpose of supporting the efficient creation of software products within a specific domain. A very important result of the Domain Engineering activity is a model describing the variability and commonality of the realizable products. This variability model<sup>1</sup> implicitly defines the scope of the product line, that is, the full range of products that are meant to be produced by it. Typically, *feature models* (Section 2.2) are used to represent variability [KCH<sup>+</sup>90, KCH<sup>+</sup>92, KLD02, CE00].

---

<sup>1</sup>Although both commonality and variability are represented in this model, we will follow the common practice of designating it as a *variability model* only, for brevity sake.

Application Engineering entails creating new products according to the specific requirements of the clients. This is achieved by taking advantage of the platform developed in the Domain Engineering step. The process involves creating a new product *configuration* (Section 2.2.2), which describes, in conformity with the variability model, the features of the desired product. This configuration is the basis for the selection and ensuing composition of the reusable assets in order to create the new product. Although ideally no additional development effort should be required, it may be necessary to address specific functionalities or properties that are not directly achievable via composition of the reusable assets. The Application Engineer must then create the *application assets* containing application-specific content. Still, the platform is expected to provide the means for efficiently creating the product, so additional product development effort beyond configuration and composition of the base assets should be reduced. Application Engineering provides crucial feedback to Domain Engineering in the form of suggestions to develop support for new features. This process tends to migrate application assets into the core domain assets, so that they can be reused if necessary in new products or newer versions of the same product. In this way, the scope of the SPL evolves to accompany the needs of the clients.

## 2.2 Feature Models

Feature models are frequently used in SPL development for identifying configurations corresponding to products, or variants, that can be created by an application engineer using the SPL [KLD02, Cle01]. Indeed, feature models identify valid product configurations by using a feature tree annotated with additional domain constraints. These can be represented graphically (e.g., linking dependent features with a dependency arrow) or by textual annotations. In a SPL, products are characterized by the features they include. Typically, a feature modularizes an increment in functionality [KCH<sup>+</sup>90, Bat06], although non-functional features may also be considered [KKL<sup>+</sup>98]. The presence of some features may be required in all products generated by the SPL, while the inclusion of other features may be optionally determined according to the specific needs of the client. Additional constraints, depending on the domain, must also be respected to ensure feasibility of the product. Feature models are visual representations of such commonality and variability in SPLs. They define the scope of the SPL by describing what products (that is, specific configurations of features) are admissible.

Basic feature models have been first described by Kang et. al in the Feature Oriented Domain Analysis (FODA) report [KCH<sup>+</sup>90]. An hierarchical tree model is used to represent mandatory, optional, and alternative relations between features and their children. Additional constraints specifying crosstree dependencies can also be used, specifying the need for mutual exclusion or inclusion between pairs of features. Later works allowed the generalization of constraints to arbitrary boolean propositions [Bat05]. Basic feature models have found widespread acceptance since their introduction, having been used in the context of generative programming [CE00], feature-oriented programming [Bat06], software factories [GSCK04] and model driven development [TBD07].

Extension and generalizations of basic feature models have been proposed. Cardinality-based feature models generalize feature models by allowing varying degrees of multiplicity when describing feature dependencies [CHE05, RBSP02]. These allow a feature to be selected multiple times, rather than being just included or excluded from a product. Feature cardinality is an interval  $[m, n]$  that describes the number of times a child feature may be selected if its parent is also selected, where  $m$  is the lower bound and  $n$  the upper bound. For example, an optional feature in the FODA model can be described as having cardinality  $[0, 1]$ , while a mandatory feature can be described using the cardinality range  $[1, 1]$ . Similarly, group cardinality dictates the number of children features that must be selected in each alternative group.

In extended (or *attributed*, or *advanced*) feature models, features include attributes that provide additional information [KKL<sup>+</sup>98, Bat05, BBRC06, BTRC05, SRI03]. These attributes are not restricted to boolean values, and represent important qualities such as cost or performance metrics, thereby allowing constraints of increased complexity to be expressed. Some configuration description languages originating from the operating systems domain such as kconfig [ZC] or CDL [VD01] allow the textual representation of extended feature model-like structures, although options like embedded script coding can go beyond the scope of what is representable in feature models.

Since they were originally introduced, feature models have found applications in many different domains such as telecom systems [GFD98, LKL02], template libraries [CE00], web services [RF03, LMN08], networking protocols [BB02], home automation [MRP<sup>+</sup>07], context-aware and mobile systems [MAW11] and embedded systems [CBUE02, LSPS05, Beu01]. A detailed survey of feature model variants and their formalization can be found in [SHTB06], while a literature review of automated analysis and reasoning can be found in [BSRC10].

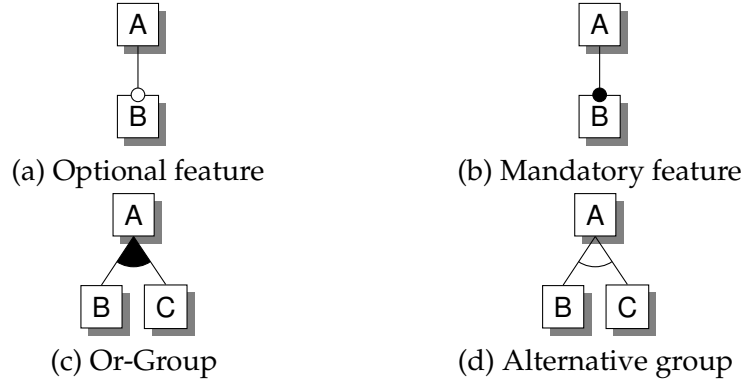


Figure 2.1: Feature tree elements

Regardless of the specific type of feature model being considered, the challenges outlined in Chapter 1 are always relevant. Nevertheless, a specific type of feature model must be considered when devising the required configuration support algorithms. Due to their widespread acceptance and use, our work is therefore based on basic feature models. Whenever we refer to feature models, in the sequel, we refer to basic feature models unless otherwise noted.

Figure 2.1 describes the graphical elements commonly used to construct feature models. In Figure 2.1a, feature *A* is represented as having an optional subfeature *B*, whereas in Figure 2.1b the subfeature is mandatory. Figure 2.1c and Figure 2.1d represent groups of features, known as OR-groups or Alternative-groups. Constraints can be represented graphically or textually. Well-known transformations, described in [Bat05, CW07], can be used to convert feature trees into Boolean logic expressions. A feature model expression is obtained by joining the feature tree expression with the domain constraints.

An example of a feature model can be found in Figure 2.2 (adapted from [BSRC10]), where "Sound", "Keyboard" and "Screen" are mandatory subfeatures of the root feature node "Phone", while "MP3Player" and "Camera" are optional subfeatures. "Polyphonic" and "Monophonic" are mandatory and alternative subfeatures of the "Sound" feature, and "Monochromatic" and "Color" are alternative subfeatures of the "Screen" feature. The requires arrow represents a domain constraint describing that selection of the "Camera" feature implies the selection of the "Color" feature.

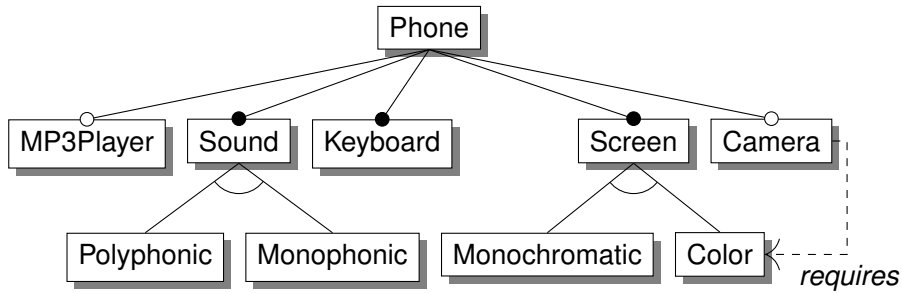


Figure 2.2: Mobile phone feature model

### 2.2.1 Boolean Logic Representation of Feature Models

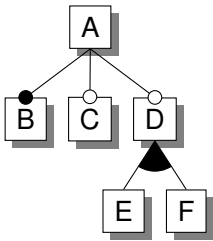
Feature models can be represented using Boolean propositional logic. Each feature is represented as a variable in the proposition and a set of well-known transformations is applied [Bat05, CW07] to obtain a Boolean proposition expression based on the topology of the feature tree and constraints. This expression will evaluate to  $\top$  (true) for all valid configurations and  $\perp$  (false) for all invalid configurations. The transformations are represented in Table 2.1. The feature model ex-

Table 2.1: Feature model transformation into Boolean propositional logic

Feature Model Element	Expression
	$\top$
	$B \Rightarrow A$
	$B \Leftrightarrow A$
	$A \Leftrightarrow (B \vee C \vee D)$
	$B \Leftrightarrow (A \wedge \neg C \wedge \neg D) \wedge$ $C \Leftrightarrow (A \wedge \neg B \wedge \neg D) \wedge$ $D \Leftrightarrow (A \wedge \neg B \wedge \neg C)$

pression is obtained by conjoining the expressions obtained from application of the transformations in Table 2.1 with all domain constraints. While the root feature can be modeled as  $\top$ , it is convenient sometimes to ignore this to achieve homogeneous treatment of all features (e.g., in feature model composition or refactoring, the root feature may change, so it can be convenient to have all features explicitly represented in the feature model expression). In this case, an additional Boolean variable corresponding to the root will appear in the feature model expression. However, this variable should be necessarily selected (most likely automatically) at the start of a new configuration process, to ensure that a void configuration is not deemed valid. Table 2.2 presents an example of the transformation of a feature model into an equivalent Boolean proposition.

Table 2.2: Feature model transformation example

Feature Model	Transformation	Root-preserving transformation
	$(\top \Leftrightarrow B) \wedge$ $(C \Rightarrow \top) \wedge$ $(D \Rightarrow \top) \wedge$ $(D \Leftrightarrow (E \vee F))$	$(A \Leftrightarrow B) \wedge$ $(C \Rightarrow A) \wedge$ $(D \Rightarrow A) \wedge$ $(D \Leftrightarrow (E \vee F))$

### 2.2.2 Feature Model Configuration

A configuration represents a specific product by describing the selected/deselected state of each available feature. A configuration is said to be *valid* if it conforms to the feature model structure and satisfies all (hard) constraints. Conformance is achieved by any configuration where:

- The root of the feature tree is selected
- All parents of selected subfeatures are selected
- All mandatory children of selected features are selected
- Exactly one subfeature is selected in each alternative group whose parent feature is selected
- One or more subfeatures are selected in each or-group whose parent feature is selected

- All domain constraints are satisfied

Useful feature models should always be *consistent*: a consistent feature model allows at least one valid configuration. An example of a valid configuration for the feature model in Figure 2.2 is:

- *Selected Features*=[Phone, Sound, Keyboard, Screen, Camera, Polyphonic, Color]
- *Deselected Features*=[MP3Player, Monophonic, Monochromatic]

A *partial configuration* is one where not all features are selected or deselected. Those features are said to be *open*.

## 2.3 Iterative Configuration of Feature Models

Iterative (or *interactive*, or *staged*) configuration of feature models is a process by which the Application Engineer creates a new product configuration by choosing to select or deselect a single open feature, in an iterative process, until all features are either selected or deselected [vdS04, Jan08, Bat05, CHE05]. Automated support for this process ensures that the choices are adequately propagated so that a valid configuration will always be obtained. To achieve this, common and dead features are automatically identified. The former are features that are always selected in all valid configurations including the current partial configuration. Conversely, the latter are features that are always deselected in the same configuration. By automatically selecting common features and deselecting dead features after each configuration action of the user, configurators ensure that a valid solution is always obtained, as long as the feature model is not void. Both Binary Decision Diagrams (BDD) [Bry86, vdS04] and satisfiability (SAT) analysis [Jan08, Bat05] have been used for configuration of feature models.

BDD-based configurators identify common and dead features by verifying if nodes are always selected or deselected in all possible paths from the root down to the  $\top$  leaf node.

The BDD of an expression  $f(x_0, x_1, \dots, x_i, \dots)$  is created by selecting some variable  $x_i$  and recursively obtaining two BDD sub-trees corresponding to the evaluation of  $f(x_0, x_1, \dots, x_i, \dots)$  for  $x_i = \top$  and  $x_i = \perp$ . Additional optimizations, such as merging identical subtrees, are also performed to improve compactness of the tree. The size of a BDD is dependent on the order by which variables are recursively selected at each node. Although creation of optimal BDDs is an NP-hard problem [BW96], efficient heuristics are known [EFD05].

An example is provided in Figure 2.3. On the left side of the figure, a BDD

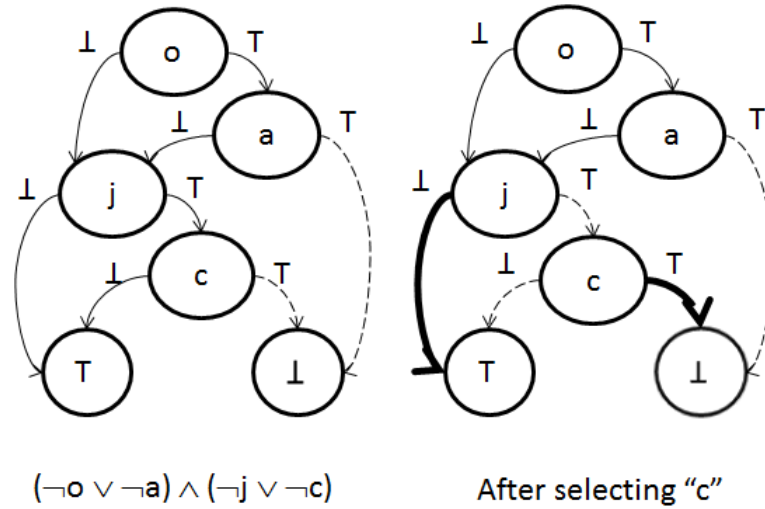


Figure 2.3: Example of BDD and its use for configuration purposes.

corresponding to the expression of the feature model in Figure 2.4 is shown (refer to the figure for the variable identification key). Starting from the top and pro-

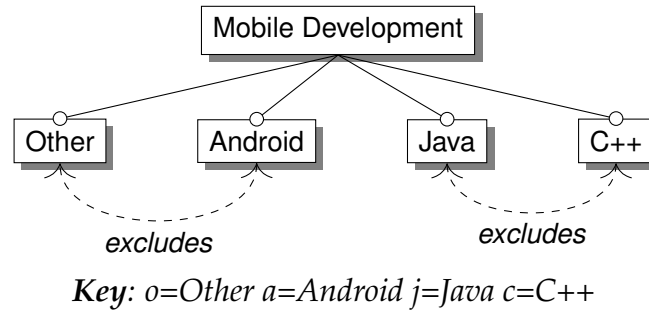


Figure 2.4: Feature model for configuration example

ceeding downwards according to the values assigned to the variables "a", "o", "j" and "c", a path is created running down from the root to one of the leaf nodes corresponding to the value of the function for that variable assignment. Dashed paths arcs do not lead to a  $\top$  evaluation so they may not be transversed in a path from root to  $\top$ , in the context of a configuration process. The right side of the figure displays the use of the BDD when the feature "c" has just been selected. This action makes the path from "c" to  $\top$  unavailable, which in turn means that no path exists from root to  $\top$  that uses the arc going from "j" down to "c". All such paths must now transverse the arc going from node "j" to node  $\top$ . This arc corresponds to "j"= $\perp$ , so "j" is identified as a dead feature that must be deselected.

SAT analysis can be used to determine dead or common features by assessing

the satisfiability of certain expressions. For example, given the feature model expression  $F(a, o, j, c)$ , if  $F(a, o, j, c) \wedge a$  is not satisfiable then "a" is a dead feature. Conversely, if  $\neg \text{SAT}(F(a, o, j, c) \wedge \neg a)$ , then  $a$  is a common feature. Since SAT-solvers can typically provide a variable assignment that satisfies the expression under analysis (if it is found to be satisfiable), then multiple results can be obtained with a single pass of SAT analysis. For example, if a variable "x" is selected in such an assignment, then the same assignment also satisfies  $F(\dots) \wedge x$ .

Referring to the same scenario that is described in example in Figure 2.3, after selection of "c", the possible configurations can be described by  $F(a, o, j, c) \wedge c$ . The configurator will proceed by trying to identify common and dead features in the remaining unassigned variables (also referred to as *open features*). Beginning, for example, with feature "a", it will evaluate  $\text{SAT}(F(a, o, j, c) \wedge c \wedge a)$  to verify if it is dead. However, this expression is satisfiable by the assignment  $\{j=\perp, o=\perp, c=\top, a=\top\}$ , so that is not the case. Since  $j=\perp$  in that expression, the same result also satisfies  $F(a, o, j, c) \wedge c \wedge a \wedge \neg j$  and, consequently  $F(a, o, j, c) \wedge c \wedge \neg j$ , so "j" can be found NOT to be common by the same result. Similar analysis can be made for "o". So, in a single pass of the SAT solver, the configurator is able to determine that "a" is not dead, "j" is not common, and "o" is not common. Next, a new pass of the SAT solver would be used to assess if "a" is common, by computing  $\text{SAT}(F(a, o, j, c) \wedge c \wedge \neg a)$ . The satisfying assignment  $\{a=\perp, o=\top, j=\perp, c=\top\}$  would be found, from which it could be further determined that "a" is not common, "o" is not dead (and also that "j" is not common, though that is already established). "o" and "a" have already been found not to be neither dead nor common, so all that is left is to verify if "j" is dead. In fact,  $F(a, o, j, c) \wedge c \wedge j$  is not satisfiable, so "j" is indeed a dead feature, and can be deselected automatically. Other approaches to configuration can also be found in the literature. Knowledge-based configuration is an area of artificial intelligence where techniques such as expert systems or constraint programming are used for configuration purposes [Stu97]. The approach entails creating a knowledge base by a Knowledge Engineer, that is then processed using one of the referred techniques. Configuration knowledge bases are often built using rules-based approach [BOBS89] or domain-specific ontologies [STMS98, FFJ01], rather than relying specifically on feature models as is our case. In this case, the knowledge base plays a role similar to that of the feature model, in that it is expected to describe valid configurations (or the actions required to achieve a valid configuration, in the case of rule-based systems).

## 2.4 Assisting the Decision Process

Regardless of the configuration technique used, the aim of the previous approaches is to steer the user into creating valid configurations. This is understandable, as all approaches are primarily based on information that describes only what valid configurations are. A different matter is helping the user to make decisions in the presence of multiple valid alternatives. Recommender systems play such a role, and are used for advising a user who is selecting one product from a set of pre-existing alternatives [FJN<sup>+</sup>13, JZFF10, RRSK11], using both historical data and domain-related heuristics. However, they do not address the issue of variable configuration of products, as they are concerned only with orienting the user in choosing one option from among a set of pre-existing solutions. The probabilistic feature models framework, described in [CSW08], is capable of orienting the user, during the configuration process, towards solutions that are correlated, from a frequency perspective, to the current partial configuration. While this is a most useful capability, dependencies between features that are not strictly related to frequency data are not easily addressed.

Some approaches are based on identifying a configuration that is optimal according to some criteria [NBD14, BDNRG10]. These approaches suggest a complete configuration to the stakeholder, rather than providing suggestions during the configuration process. Typically, only one "ideal" configuration is identified even if multiple alternatives exist. No stakeholder input is sought when considering what to do when multiple alternative and mutually exclusive configurations exist. In [NBD14], support is offered for the ideation step (see Chapter 1) by identifying desirable features, according to a goal model describing user's intentions and wishes, by performing semantic analysis of feature model documentation. All this takes place before the actual configuration (i.e., the realization step), and there is no concern about validity. So, according to our model presented in Figure 1.1, this work supports construction of the idealized configuration. It does not consider potential conflicts, since feature model constraints are not taken into account. This approach complements ours, since in our work we offer support during the ensuing realization process, by identifying required trade-offs necessary for ensuring validity and providing suggestions that might have not been considered during the original ideation process. In [BDNRG10], fuzzy reasoning is used to determine the configuration that maximizes the number of satisfied fuzzy soft constraints. No consideration is made on the existence of alternative optimal configurations or potential trade-offs.

The C20 configurator is presented in [NE13]. It is based on decision models, that describe questions, possible answers, constraints and relevancy relations between those questions. The configurator supports the decision-making process, by guiding the user so that the number of questions to be answered is optimized via an heuristic process [NE11], but allows the user to deviate and explore inconsistent solutions, which must be fixed later in the configuration process. Although we do not attempt to minimize the number of required configuration actions, this work addresses other concerns similar to our own, such as configuration-order issues and inconsistency handling, but there are some outstanding differences. In C20, soft constraints are not considered, so no domain-related advice is provided that helps addressing sub-optimal ideations. Choices are therefore always immediately propagated according to constraints and relevancy relations. These choices may be later changed by the user, entering an inconsistent state, that must be necessarily fixed in the sequel. We provide advice to the user that steers towards satisfaction of soft constraints and preemptively identifies choices that must be made to avoid conflicting features, but do not propose the automated change of already configured features: either the conflict is inevitable or the user has deliberately decided to go against the configuration advice at an earlier point. Rather than allowing the configuration to enter an inconsistent state, we allow for inconsistent specifications to be handled by upfront modeling of user goals as a set of (possibly inconsistent) soft constraints, or by resorting to prototype-based configuration (see Chapter 6), where a complete (or partially complete) configuration can be specified as a single configuration step, which is then repaired if necessary to ensure validity is preserved.

Explanation-providing systems focus on identifying minimal sets of changes required for satisfying an overconstrained specification [Jun04, Rei87]. These approaches are primarily based on the identification of minimal unsatisfiable cores. A minimal unsatisfiable core of a Boolean proposition in conjunctive normal form is a set of minimal dimension including the clauses that cannot be successfully satisfied (e.g.,  $\neg A \wedge \neg B \wedge (A \vee B)$ ). While many approaches for computing minimal unsatisfiable cores exist [LS04], identification of the *minimum* (globally optimal) unsatisfiable core can be challenging. It can also be hard for a stakeholder to trace back clause expressions to domain constraints and feature tree structure.



# Boolean Soft Constraints

A part of our proposed solution is the configuration advisor that offers additional support to the stakeholder during the configuration process. As illustrated in Figure 1.2, this advisor requires the use of soft constraints (SC) to represent domain information or stakeholders' requirements. This chapter discusses different types of soft constraints applied to feature models, and investigates which semantics are most suitable for the purposes of our work.

We begin by presenting related work concerned with the use of soft constraints in feature modeling (Section 3.1). Our original contributions discussed here are the Boolean soft constraints for feature models, the categorization of their semantics as normative or annotational (Section 3.2), and the description of a framework enabling the use of normative constraints in feature models (Section 3.3). The role of annotational semantics is also discussed (Section 3.4). The use of soft constraints in feature modeling relates with research question 1, while using soft constraints to model overconstrained scenarios relates to research question 3a.

## 3.1 Soft Constraints in Feature Modeling

Soft constraints have been largely ignored in SPL development until recent years. Early efforts include [RP03], where a fuzzy logic description of customer profiles

and domain constraints is added to feature models. "Encourages" and "discourages" constraints have been proposed for feature models in [WSO07]. However, these are used only to generate graphical decorations of the feature model: no precise semantics have been provided, precluding automated analysis and reasoning as described in our work.

In [CSW08], Czarnecki et al. describe probabilistic feature models. In these, features can be related by a probability space (PSPACE) that describes probabilities of combined selection of features. This can be used to generate advice for steering the user into commonly selected feature configurations. Probabilistic feature models are well suited for representing the outcome of a feature mining process, by accurately representing observed frequencies. However, it can be challenging to represent domain information that is not directly related to frequency information (e.g., a suggestion to select a newly introduced feature).

In [BDNRG10], Bagheri et al. describe the use of fuzzy-based soft constraints to model stakeholder's preferred configurations. A reasoning algorithm identifies the optimal configuration that maximizes constraint satisfaction. This approach differs from ours because it is concerned only with maximal constraint satisfaction. It does not address the question of trade-off identification and explanation nor is it concerned with the exploration of multiple alternative optimal solutions.

## 3.2 Boolean Soft Constraints in Feature Models

As a first step towards discussing the use of SCs in feature models, we first focused our efforts in addressing the nature of soft constraints and explored possible alternative semantics [BM11]. From early on, we decided to focus our efforts on Boolean soft constraints, even though fuzzy logic is more commonly considered in the context of handling uncertainty and soft goals. This decision is based on the following considerations:

1. Standard feature models can be readily formalized as Boolean expressions [Bat05]. By using the same formalism, we can leverage a large body of research and tools already available.
2. The hard or soft nature of constraints is not directly related to the specific logic formalism that is used, but with the mandatory or optional nature of constraint satisfaction.

3. The consequences and impact on results of the parametrization of a fuzzy-logic approach (e.g., fuzzification/defuzzification, selection of fuzzy operators, etc.) may prove to be challenging to anticipate.
4. Other works [RP03, BDNRG10] had already begun exploring fuzzy-based approaches, so by exploring alternative solutions we believe a greater contribution may be offered.

Boolean SCs are described in Table 3.1. They are represented by using one of four different binary operators applied to two Boolean propositions  $P$  and  $Q$ .  $P$  is said to be the *trigger* of the soft constraint, while  $Q$  is the *consequence*. Although for completeness four different operators are represented, all constraints can be rewritten using only a single operator, as indicated in the second column of the table. Therefore, in the remaining text, we usually refer only to the first operator (*suggests*).

Table 3.1: Boolean soft constraints

Soft Constraint	Equivalent to
$P$ suggests $Q$	-
$P$ discourages $Q$	$P$ suggests $\neg Q$
$P$ absence-suggests $Q$	$\neg P$ suggests $Q$
$P$ absence-discourages $Q$	$\neg P$ suggests $\neg Q$

The following criterium is used to determine satisfaction of a soft constraint

The soft constraint  $P$  suggests  $Q$  is said to be *satisfied* iff  $P \Rightarrow Q$

A constraint is said to be *triggered* if its trigger  $P$  is  $\top$  (true). Conversely, a constraint is *untriggered* if  $P$  is  $\perp$  (false). A constraint is *activated* if  $Q$  is  $\top$ , and *inactivated* otherwise.

Owing to the non-mandatory nature of SCs, we began our investigation by trying to understand possible semantics for soft constraints, when applied to feature models. Valid feature configurations comply with all hard constraints and the feature tree structure. Within the set of all these configurations, we will typically be able to find a variable degree of SCs satisfaction, ranging, in the general case, from complete disregard for soft constraints (no SCs are satisfied) up to full compliance (all SCs are satisfied). Considering a colloquial interpretation of soft constraints that prescribes them as something that "should be satisfied, if possible", one may wonder if such configurations that are not optimal in the sense of soft constraint satisfaction should be considered admissible or acceptable (in

other words, actually valid). Depending on the answer to that question, we identified two different semantic alternatives for soft constraints in feature models:

- **Normative Semantics.** This is a strict interpretation of the satisfaction requirement for soft constraints — if it is possible to do so without compromising hard constraints, soft constraints must be satisfied.
- **Annotational Semantics.** Conversely, annotational semantics impose no such requirement. All configurations conforming to the hard constraints and feature tree are considered equally valid, albeit some may be preferable to others.

This distinction is important because normative semantics impose additional restrictions on the validity of the solutions. That is not the case for annotational semantics.

### 3.3 Normative Semantics

A normative soft constraint must be considered when assessing the validity of a product configuration. These constraints represent configuration information that may potentially condition the validity of some configurations. A normative soft constraint must be satisfied if possible, but can be ignored otherwise. The concept of "possible satisfaction" must be further clarified, since unless the constraint is impossible to satisfy in all configurations, it is obviously possible to change the configuration in some way that will satisfy it. Furthermore, any changes to a configuration might have implications in the satisfiability of other soft constraints, suggesting that a priority-based ordering of normative soft constraints may be required. In [BM11], we formalize the impact of normative soft constraints of the format  $A(f_1, \dots)$  suggests  $f_m$  in the validity of a configuration, where  $A(f_1, \dots)$  is a Boolean proposition and  $f_m$  a Boolean variable. This is achieved by recursively adapting the feature model expression, initially obtained by the transformations described in Section 2.2.1, to account for the introduction of each normative soft constraint, in priority order. This can be generalized as follows. Consider the introduction of a single normative constraint  $A(f_1, \dots)$  suggests  $B(f_1, \dots)$  in a feature model with expression  $F_0(f_1, \dots)$ . The impact of the new constraint can be represented by finding a new feature model expression  $F_1(f_1, \dots)$  where

$$F_1(f_1, \dots) = F_0(f_1, \dots) \wedge \left( (A_0(f_1, \dots) \Rightarrow B_0(f_1, \dots)) \vee I_0(f_1, \dots) \right) \quad (3.1)$$

and  $I_0(f_1, \dots)$  is a logic function that denotes the *impossibility* of satisfying the constraint  $A_0(f_1, \dots) \Rightarrow B_0(f_1, \dots)$ , or equivalently if it is acceptable not to satisfy it (we will describe the nature of this function and possible alternatives in the sequel). Equation 3.1 indicates that a configuration is valid after the introduction of the new constraint if it was previously valid and either the new constraint is satisfied or it is impossible (or acceptable not) to satisfy it. Accordingly, the effect in validity of the introduction of the  $n$ -th highest priority constraint  $F_n(f_1, \dots)$  can be described similarly in terms of  $F_{n-1}(f_1, \dots)$ .

$$F_n(f_1, \dots) = F_{n-1}(f_1, \dots) \wedge \left( (A_{n-1}(f_1, \dots) \Rightarrow B_{n-1}(f_1, \dots)) \vee I_{n-1}(f_1, \dots) \right) \quad (3.2)$$

Different functions can be considered for assessing the possibility of changing the configuration in order to satisfy the constraint  $A_{n-1}(f_1, \dots) \Rightarrow B_{n-1}(f_1, \dots)$ . Those functions reflect potential changes to the configuration and the structure of the constraint, as well as the structure of the feature model and other constraints. This means that typically  $I_n(f_1, \dots)$  will be dependent on  $F_n(f_1, \dots)$ .

In [BM11], we consider constraints of the format  $A(f_1, \dots)$  *suggests*  $f_m$  (or  $A(f_1, \dots)$  *suggests*  $\neg f_m$ ). If such a constraint is not satisfied, one possibility for achieving satisfaction is toggling  $f_m$ . The "possibility" of satisfying a currently unsatisfied soft constraint is therefore determined according to the following criterium: a configuration that does not satisfy a constraint  $A(f_1, \dots)$  *suggests*  $f_m$  is only valid if toggling the selection state of  $f_m$  would otherwise make it invalid. This would happen if the resulting configuration failed to satisfy a hard constraint, the feature tree structure or a higher priority soft constraint is not satisfied when that would be possible. For formalizing this criterium, we can use the following impossibility function

$$I_n = \neg F_n(f_1, \dots, \neg f_m, \dots) \quad (3.3)$$

This simply states that it is not possible to achieve satisfaction of  $A(f_1, \dots)$  *suggests*  $f_m$  by toggling  $f_m$  if the result is an inviable configuration according to feature tree, hard constraints and higher priority SCs.

Figure 3.1 presents a feature model with two normative soft constraints: "Android" *suggests* "Java" and "Mobile Development" *discourages* "C++". Other constraints are hard. In this example, we wish the constraint "Android" *suggests*

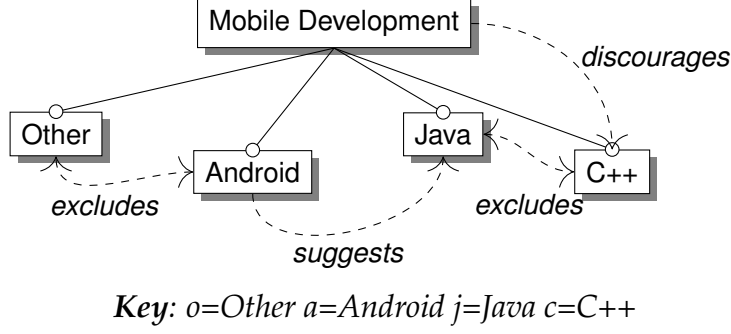


Figure 3.1: Normative soft constraints example (the key describes the condensed variable notation used in the text)

"Java" to have the highest priority. Applying equations 3.2 and 3.3, we find that<sup>1</sup>:

$$\begin{aligned}
 F_0(o, a, j, c) &= (\neg o \vee \neg a) \wedge (\neg j \vee \neg c) \\
 F_1(o, a, j, c) &= F_0(o, a, j, c) \wedge ((a \Rightarrow j) \vee \neg F_0(o, a, \neg j, c)) \\
 F_2(o, a, j, c) &= F_1(o, a, j, c) \wedge ((\neg c \Rightarrow \neg j) \vee \neg F_1(o, a, j, \neg c)) \\
 &= \dots \\
 &= a \wedge c \wedge \neg j \wedge \neg o \vee \neg c \wedge j \wedge \neg o \vee \neg a \wedge \neg c
 \end{aligned} \tag{3.4}$$

Equation (3.4) describes valid configurations that conform to the feature model and both normative soft constraints. Results are presented in tabular form in Table 3.2. It can be seen that configurations such as  $\{\neg a, c, \neg j, \neg o\}$  or  $\{a, \neg c, \neg j, \neg o\}$  are not considered valid. In both cases, this is due to the normative constraints, since it is possible, according to the definition in Equation 3.3, to satisfy each constraint by toggling the configuration state of a single feature. In the first case, "Mobile Development" *discourages* "C++" is not satisfied, but could be by toggling selection of the "C++" feature, without compromising conformity to the feature model or satisfaction of the high priority constraint. Similar considerations apply to the latter case, with respect to the "Android" *suggests* "Java" constraint and toggling of the "Java" feature.

Table 3.2: Normative constraint example results

	$\neg j \neg o$	$\neg j o$	$j o$	$j \neg o$
$\neg a \neg c$	1	1	1	1
$\neg a c$	0	0	0	0
$a c$	1	0	0	0
$a \neg c$	0	0	0	1

<sup>1</sup>Please refer to the key in Figure 3.1 to associate variables with the corresponding features.

One interesting case is the configuration  $\{a, c, \neg j, \neg o\}$ . It fails to satisfy both SCs, but is nevertheless considered valid. This happens because Equation 3.3 is obtained by considering possible changes to the configuration involving only a *single* feature. However, it is not possible to make any change to a single feature in  $\{a, c, \neg j, \neg o\}$  that results in a valid configuration satisfying either one of the constraints. This can be easily observed in the Karnaugh map [Kar53] presented in Table 3.2, where it is apparent that no valid configuration adjacent to  $\{a, c, \neg j, \neg o\}$  exists. If one considers toggling the "Java" feature to satisfy the high priority constraint, the resulting configuration will be invalid due to the mutual exclusion of "Java" and "C++". On the other hand, toggling off "C++" will result in a configuration with neither "Java" nor "C++" selected. In such a configuration, the normative constraint "Android" suggests "Java" is not respected, since it would be possible to toggle the "Java" constraint without impact on validity. This would, however, require toggling two features ("Java" and "C++") simultaneously, and that option is not contemplated in Equation 3.3. This situation is due to the relatively restrictive assumption about permissible changes to the configuration that was considered when developing Equation 3.3. However, alternative impossibility functions can be considered, allowing far more versatile changes to be contemplated.

### 3.3.1 On Impossibility Functions

An impossibility function  $I_x(f_1, \dots)$  can be interpreted as describing the conditions under which it is considered to be acceptable NOT satisfying a certain constraint  $A_x(f_1, \dots) \Rightarrow B_x(f_1, \dots)$ . For example, if  $I_x(f_1, \dots) = \top$ , then it is always acceptable not to satisfy the constraint. Conversely, if  $I_x(f_1, \dots) = \perp$ , it is never acceptable not to satisfy it. It becomes, in fact, a hard constraint. In between these two extremes, alternative formulations can be created to reflect intermediate scenarios where acceptance of unsatisfied constraints is contingent on specific conditions. Considering  $F_n(f_1, \dots)$  represents the validity of a configuration in a certain feature model with  $n$  higher priority constraints applied, then Equation 3.3 can be understood as determining that it is acceptable not to satisfy the associated constraint only if toggling a certain feature, that would make it satisfiable, would make the configuration otherwise invalid. It is simple to generalize this approach to more elaborated conditions.

Any unsatisfied constraint  $A_x(f_1, \dots) \Rightarrow B_x(f_1, \dots)$  can be satisfied by any change in one or more variables  $f_i$  that either makes  $A_x(f_1, \dots)$  false or  $B_x(f_1, \dots)$  true (or both). For example, the unsatisfied constraint *a suggests b* can be satisfied

either by making  $a = \perp$  or  $b = \top$ . Consider now the model in Figure 3.2 and a more complex constraint with  $A_x(f_1, \dots, f_{12}) = f_5 \wedge f_7$  and  $B_x(f_1, \dots, f_{12}) = f_8 \vee f_9$  applying to that model, resulting in a suggestion to use either the Eclipse or Netbeans tools for in-house development of Android applications. Consider now that it is deemed unacceptable failing to satisfy this constraint if any one of the following options is possible without hindering the satisfaction of higher priority constraints or conformity to the feature model:

1. Using the Eclipse ( $f_8$ ) tool.
2. Using the Netbeans ( $f_9$ ) tool.
3. Resorting to outsourcing ( $f_4$ ).

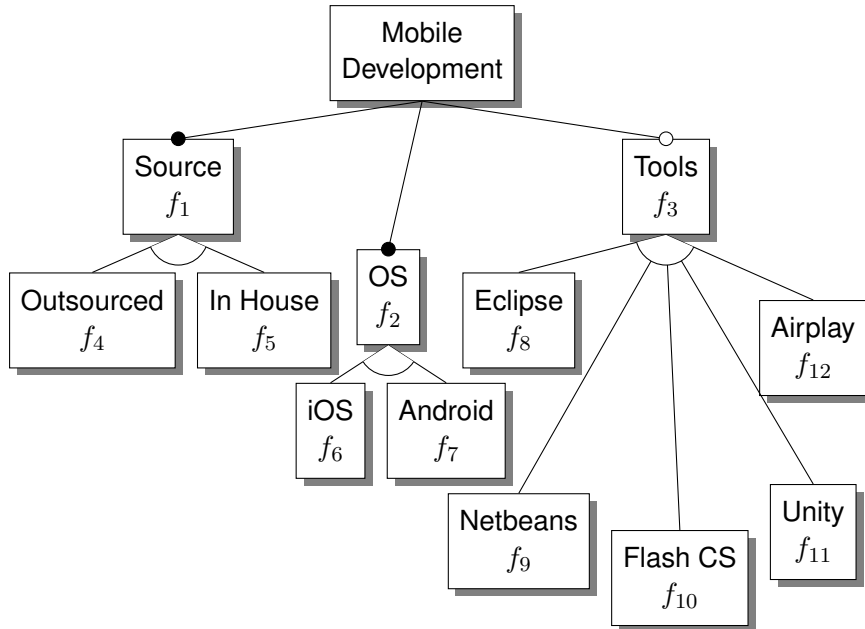


Figure 3.2: Complex Normative Constraint Example

Each such option ensures satisfaction of the constraint by either making  $A_x(f_1, \dots, f_{12})$  false (option 3) or making  $B_x(f_1, \dots, f_{12})$  true (options 1 and 2). Whether or not an order of preference between these options exists is irrelevant for our discussion. Also notice that not all conceptually possible options need to be included. In this case, it would be possible to satisfy the constraint by deselecting the "Android" feature, but that is not contemplated here. This can correspond to a domain-related concern that maintaining use of the "Android" operating system is more important than satisfying this soft constraint.

Looking at the feature model, it is clear that, regardless of other constraints, the first option requires not only that the "Eclipse" ( $f_8$ ) feature is selected, but

also that "Tools" ( $f_3$ ) is selected and all other alternative tools are deselected ( $f_9, \dots, f_{12}$ ). Similar considerations apply to the second and third options. Considering  $F_{x-1}(f_1, \dots, f_{12})$  represents consistency wrt the feature model and higher priority constraints, we find that the effect of each option may then be modeled correspondingly as:

1.  $O_1(f_1, \dots) = F_{x-1}(f_1, f_2, \top, f_4, f_5, f_6, f_7, \top, \perp, \perp, \perp, \perp)$
2.  $O_2(f_1, \dots) = F_{x-1}(f_1, f_2, \top, f_4, f_5, f_6, f_7, \perp, \top, \perp, \perp, \perp)$
3.  $O_3(f_1, \dots) = F_{x-1}(\top, f_2, f_3, \top, \perp, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{12})$

If any one of  $O_i(f_1, \dots)$  evaluates to  $\top$ , then the corresponding option may be used to satisfy the constraint. The impossibility function can be derived from these functions, when integrating the constraints via Equation 3.2. Considering

$$\begin{aligned} c_0 &= \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}\} \\ c_1 &= \{f_1, f_2, \top, f_4, f_5, f_6, f_7, \top, \perp, \perp, \perp, \perp\} \\ c_2 &= \{f_1, f_2, \top, f_4, f_5, f_6, f_7, \perp, \top, \perp, \perp, \perp\} \\ c_3 &= \{\top, f_2, f_3, \top, \perp, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}\} \end{aligned}$$

and continuing the example, if  $S_0 \dots S_{x-1}$  are the expressions satisfying soft constraints of higher priority than  $S_x$  (with  $S_x(f_1, f_2, \dots) = A_x(f_1, f_2, \dots) \Rightarrow B_x(f_1, f_2, \dots)$ ), then  $I_x(f_1, \dots)$  can be computed as:

$$I_x(c_0) = \neg \left( \bigvee_{i=1 \dots 3} (O_i(c_0) \wedge \bigwedge_{j=0 \dots x-1} (S_j(c_0) \Rightarrow S_j(c_i))) \right) \quad (3.5)$$

Equation 3.5 can be explained as follows. The inner conjunction  $\bigwedge_{j=0 \dots x-1} (S_j(c_0) \Rightarrow S_j(c_i))$  iterates over all constraints with higher priority and ensures any constraints satisfied in the original configuration are not unsatisfied after applying the changes corresponding to vector  $c_i$  (a higher priority constraint may *become* satisfied by the change, but it may not cease to be satisfied, therefore  $S_j(c_0) \Rightarrow S_j(c_i)$ ). If the inner conjunction is satisfied (satisfaction of higher priority constraints is not hindered) and it is possible to perform the corresponding change ( $O_i(c_0)$ ) then it would be possible to satisfy the constraint  $S_x(c_0)$  by making the corresponding transformation. The outer disjunction ensures this analysis is performed for all possible satisfaction options. The outer negation ensures that  $I_x(c_0)$  is  $\top$  only when none of the provided options is feasible.

### 3.3.2 Annotation of a Feature Model with Normative Soft Constraints

From the perspective of someone creating a feature model with soft constraints, most of this process can be automated so that the underlying details and mathematics remain hidden. Other than providing the feature model, the user must provide the following:

- The soft constraint description (e.g., "In House" *and* "Android" *suggests* "Eclipse" or "Netbeans") and associated priority.
- Identifying the potential options for making that constraint satisfiable. This can be achieved simply by providing the corresponding vectors  $c_i$ . These can be constructed by selecting the features to include and exclude in an attempt to satisfy the constraint. The former correspond to  $\top$  values in  $c_i$ , while the latter correspond to  $\perp$ . This can be as simple as typing in the features names or selecting them on a graphical user interface.

Provided the SCs are identified and that the vectors  $c_i$  are provided, each  $O_i(c_0) = F_x(c_i)$  and Equation 3.5 can be derived.

## 3.4 Annotational Semantics

Unlike normative SCs, annotational SCs do not have bearing on the validity of the configuration<sup>2</sup>. They are simply representations of domain knowledge and preferred configurations, but may be unsatisfied in any conditions. In this sense, they are "true" soft constraints in that they may be disregarded at user's discretion. The true value of annotational semantics lies in the additional information that they bring into the model. Although they do not have a direct impact on validity, they can be processed and analyzed by configuration tools for multiple purposes, including offering configuration suggestions, identifying and providing information about conflicts.

While conflicting normative SCs may also exist, the resolution of that conflict is hard-wired into the Equation 3.2, and depends completely on the feature

---

<sup>2</sup>Although strictly speaking annotational soft constraints can be considered just a specific type of normative constraints (i.e., those for which  $I_x(f_1, \dots) = \top$ ), their satisfaction is dependent only on feature assignment and not on other constraints. This is not the case for normative constraints as defined by  $I_x(f_1, \dots) \neq \top$ . When we speak of normative constraints, we therefore generally mean constraints of the latter form unless otherwise stated.

model, other SCs and their priority. In this sense, all necessary trade-offs (related to SC satisfaction) are resolved upfront, when satisfying options and priorities are determined for each constraint at the time of creating and annotating the feature model. However, it may be the case that it is difficult to assign priorities meaningfully upfront. For example, the relative importance and priority of some constraints may depend on additional factors (e.g., a constraint promoting a small memory footprint is more relevant for a mobile application than for a desktop application). Also, the preferred resolution for a certain conflict may depend completely on stakeholder preferences, thereby making it very difficult to anticipate the best resolutions. This means that not all soft constraints may be comfortably expressed using normative semantics (with  $I_x(f_1, \dots)$  given by Equation 3.5). Annotational semantics are a better solution if one has the intention of allowing the stakeholder to explicitly decide on how to resolve each conflict, because they allow conflicts to be explicitly identified but do not impose any specific type of resolution.

Since our work is concerned with enabling a user-centric perspective of product derivation, annotational soft constraints are then a better match for the type of support we envision. In the remainder of the text, therefore, we always assume an annotational interpretation of soft constraints, unless otherwise stated.

## 3.5 Conclusions

Boolean soft constraints not only allow uncertain and overconstrained user preferences to be represented, but also improve domain modeling capabilities (as discussed in Chapter 4). This information can be used by our configuration advisor (examined in Chapter 5) to provide feedback to the user during the configuration process to better allow him to realize his vision.

We described two alternative semantics for soft constraints applied to a feature model. Normative soft constraints must be satisfied "if possible", while annotational soft constraints are mostly informative in their nature and can always be disregarded. The concept of "possibility of satisfaction" is variable and can be defined on a per-constraint basis using the framework provided in this chapter. Normative soft constraints are organized in total priority order, and then combined into an aggregated constraint that is added to the feature model. Subsequent use of any standard iterative configurator (discussed in Section 2.3) is ensured to generate a configuration that will always satisfy the normative soft constraints, if possible. We also discuss specification of normative constraints from

the perspective of a domain engineer and stakeholder, and show that they can be used without deeper understanding of the underlying mathematics and framework. Resolution of conflicting normative soft constraints is implicitly achieved upfront by the definition of the priority order and definition of satisfaction possibility.

Annotational semantics are informative in nature. While they could be represented in the normative framework, configuration of such a model by an incremental configurator would be no different from configuration of the non-annotated model. Although these constraints do not directly impact the configuration process (unlike normative constraints), they represent valuable information that can be used for other purposes. Specifically, in our case, we use annotational soft constraints during the configuration process to generate advice providing configuration suggestions. We also identify conflicts and point out consequences of alternative resolutions of those conflicts. The fact that these conflicts are not resolved upfront, as is the case with normative soft constraints, is an advantage. Resolution of conflicts that actually manifest during configuration of a product can be resolved by the stakeholder directly, using the provided information for guidance and having in mind the specific context of the product being created. For this reason, annotational semantics are more suited for the purposes of our work.

# 4

## Soft Constraints in Domain Engineering

Soft constraints are used in our work to represent both domain information and stakeholder preferences, and these fuel the configuration advisor that provides an enhanced configuration experience to the user (see Figure 1.2). During domain engineering, a feature model is created that accurately represents domain information and defines the scope of the product line. It is at this time that domain-related soft constraints are introduced and integrated into the feature model.

This chapter discusses the use of soft constraints in domain engineering (Section 4.1), directly addressing research questions 1a and 1(a)i. Original contributions include the proposal of some prototypical patterns of application (Section 4.2) and use of soft constraints in feature model evolution, as well as the classification of suspicious soft constraint interactions and automated techniques for detecting those (Section 4.3).

### 4.1 Domain-related Soft Constraints

While traditional feature models include only hard constraints, soft constraints can be used to represent relations between the features that may be desirable but are not mandatory. Conflicting requirements can also be represented by soft constraints. For example, a requirement might state that "Phone models

equipped with cameras should have a color display", while another might state that "Low end models with monophonic sound should also be equipped with monochromatic screens". If "low end models" are not specifically precluded from being equipped with cameras, an indirect conflict arises (via mutual exclusion of monochromatic and color displays). Codifying these requirements as hard constraints implicitly resolves the conflict by ruling out configurations with both "Monophonic" sound and "Camera". While this may be appropriate, it is not the only conceivable resolution of the conflict, and by choosing that option, all other potentially interesting resolution strategies are immediately removed from consideration.

Another example of such conflicts is provided by the feature model represented in Figure 4.1 (adapted from [CSW08]). In this case, a soft constraint may indicate that vehicles destined to the US market should include automatic transmission, while another soft constraint might indicate that sport vehicles are preferably configured with manual transmission. Both suggestions are correct, although both cannot be satisfied simultaneously for a sports vehicle destined to the American market. Had both of these constraints been specified as hard constraints instead, it would have been impossible to create a valid sports vehicle configuration destined to the US market. This would be an example of a scenario where the range of admissible configurations would be unnecessarily restricted by hardcoding the conflicting knowledge into the feature model as hard constraints. An alternative solution would involve dropping one or both constraints, but this would be unfortunate as both provide useful configuration advice that is only conflicting if a specific partial configuration is produced. These examples illustrate the usefulness of soft constraints as a modeling tool for representing certain types of domain information.

If the developer is not initially aware of such conflicts, the implicit resolution and restriction of the configuration space, made by hardcoding the requirements via hard constraints, may come as an (undesirable) surprise. Conversely, if these requirements had been codified as soft constraints, no implicit resolution of the conflict has been made. Configuration tools may automatically identify the conflict, report it and explain it to the user, who will then resolve it in the way that is more suited for the specific product configuration being created. This improved configuration support is one of the benefits gained by including soft constraints in feature models. Traditional interactive configuration support tools are able to automatically propagate user choices as required to ensure validity is preserved. These dependencies stem from the feature tree topology and hard constraints. If

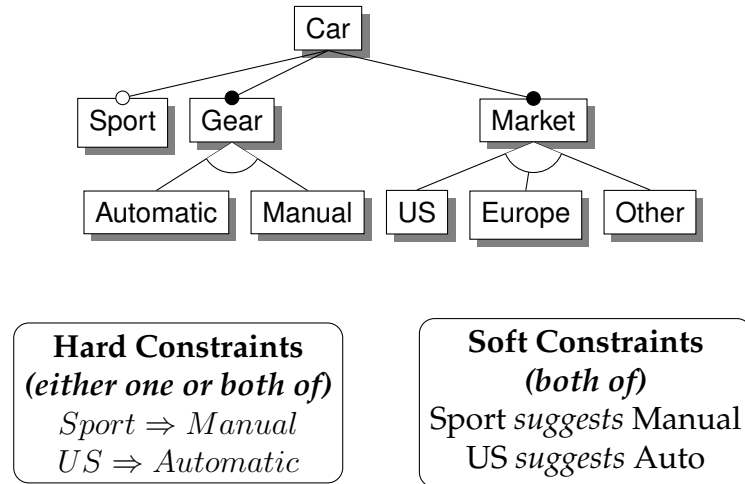


Figure 4.1: Feature model describing vehicles configuration

soft constraints are used, configuration hints or suggestions may be generated by a similar process. At each point, the configurator identifies what choices must be made to ensure soft constraint satisfaction. However, rather than automatically enforcing those choices, the user is only advised to make the corresponding choices: soft constraints are not required to be satisfied.

One important practical difference between hard and soft constraints is that conflicting hard constraints (that is, hard constraints that cannot be simultaneously satisfied) can never be found in consistent feature models. This is not possible when considering hard constraints only, otherwise the feature model would be void (would not allow any valid configuration), an anomaly that is commonly detected by feature model editing tools [KCH<sup>+</sup>92, Bat05, CHE05, BSRC10]. However, that is not the case with soft constraints since, by definition, their satisfaction is not mandatory. Conflicting soft constraints can be useful for representing domain concerns or requirements offering contradictory configuration advice. If soft constraints are not employed, resolution of such tensions must be hard-coded into the feature model, typically by identifying and dropping one of the relevant constraints or by restricting the range of admissible configurations. This may be unfortunate, as different resolutions might be preferable for different product configurations. By annotating the feature model with soft constraints, the option of deferring conflict resolution to configuration time can be made, allowing it to be done in a concrete context (i.e., knowing a partial configuration of the specific product being created) rather than up-front based on abstract considerations that may fail to apply in some concrete products.

## 4.2 Prototypical Applications

Currently, soft constraints are not commonly used in SPL development. Therefore, it is not possible to infer typical patterns of application from pre-existing case studies. Nevertheless, in this section, we point out some prototypical applications of soft constraints in domain engineering, that is, potential applications and patterns, and the rationale behind them. While this aims at providing additional insight regarding SC use, it also has found immediate practical application, since these patterns form the basis for the constraint injection algorithm used in the validation of our work (see Chapter 7).

### 4.2.1 Soft Constraint Annotation Patterns

In this section, we identify a few potential patterns for application of soft constraints. These were first described in [BM12] to support the automated creation of feature models annotated with SCs for testing purposes.

- **Optional Selection Suggestion.** A soft constraint is used to tie the configuration of separated variability options. This is a straightforward application of soft constraint semantics (Section 4.2.2).
- **Reverse Constraint Suggestion.** A soft constraint is used to strengthen the relation between dependent configuration options. It is an attempt to ensure that features are opportunistically selected or deselected provided the right conditions are met (Section 4.2.3).
- **Group Selection Suggestion.** A soft constraint is used to indicate preferential configuration options for grouped features (Section 4.2.4).

### 4.2.2 Optional Selection Suggestion

**Overview:** The pattern Optional Selection Suggestion (OSS) encompasses a broad range of situations where domain-specific interdependencies affect the configuration of optional and group children features. An OSS represents one of these situations, by interlinking the configuration of two non-mandatory features via a soft constraint. The OSS pattern represents domain-specific dependencies between features and as such has very generic scope. Specializations of this pattern may be devised if domain-specific knowledge is considered.

**Context:** A feature model describes product variability. Different features of the product may be configured independently, making it impossible to statically

correlate their configuration status via hard constraints. However, some type of domain-related dependency or preferential configuration choices can nevertheless be identified.

**Problem:** Additional domain information other than that which is captured by the hard constraints is well understood. However, the modelling tools available to the domain engineer do not allow him to represent it. As a result, this information will either be ignored or, alternatively, be represented using ill-suited modeling tools. As a consequence of the first option, it will be impossible to provide any type of automated support that requires that information. The second option leads to overconstrained models, that impose unnecessary constraints in the products being created.

**Solution:** By using soft constraints to represent domain-related preferential configuration choices, that information is not lost nor modelled using unsuitable approaches. Automated procedures such as reasoning and validation of feature configurations become possible.

**Example:** Figure 4.2 illustrates the injection of an OSS into a feature model. It suggests that if the range sensors are available, then some type of automated driving features should be included.

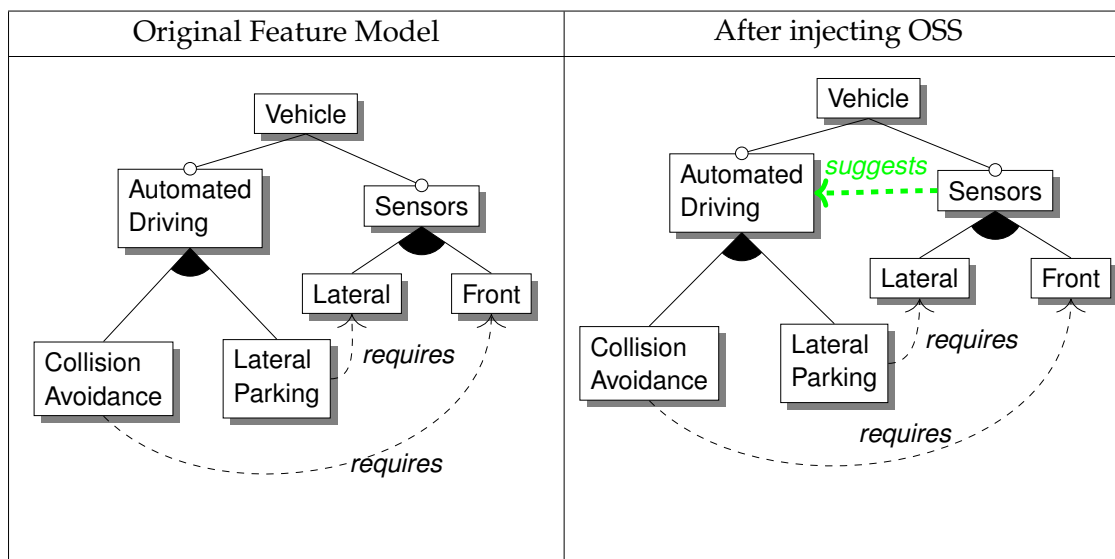


Figure 4.2: Example of Optional Selection Suggestion

### 4.2.3 Reversed Constraint Suggestion

**Overview:** The conceptual interpretation of the Reversed Constraint Suggestion (RCS) pattern is based on the intuitive notion that, in some cases, if all the requirements for a certain feature (as specified by hard constraints) are met, then it

may be sensible to opportunistically select it.

**Context:** The inclusion of a certain feature  $F$  requires that one or more other features are also selected, because the latter provide some kind of support or service required by  $F$ . Those features, however, may be included for reasons other than supporting  $F$ . The cost, or other forces that may suggest the deselection of  $F$ , are at least partially related to the inclusion of the required dependencies.

**Problem:** It can make economical sense to include the dependent feature  $F$ , if the required features dependencies are available. Not selecting  $F$ , in such a context, might result in a wasted opportunity and sub-optimal or inefficient use of available resources.

**Solution:** The RCS of constraint  $C$  is added to the model. It is a soft constraint that specifies that the reversed relation should also hold, that is,  $RCS(A \Rightarrow B) = B \text{ suggests } A$ .

**Example:** In the example in Figure 4.3, a RCS is added so that inclusion of the front range finder sensor suggests collision avoidance breaking, with the presumed rationale that this would be the way to best take advantage of the capabilities of the front range sensor, if it is included.

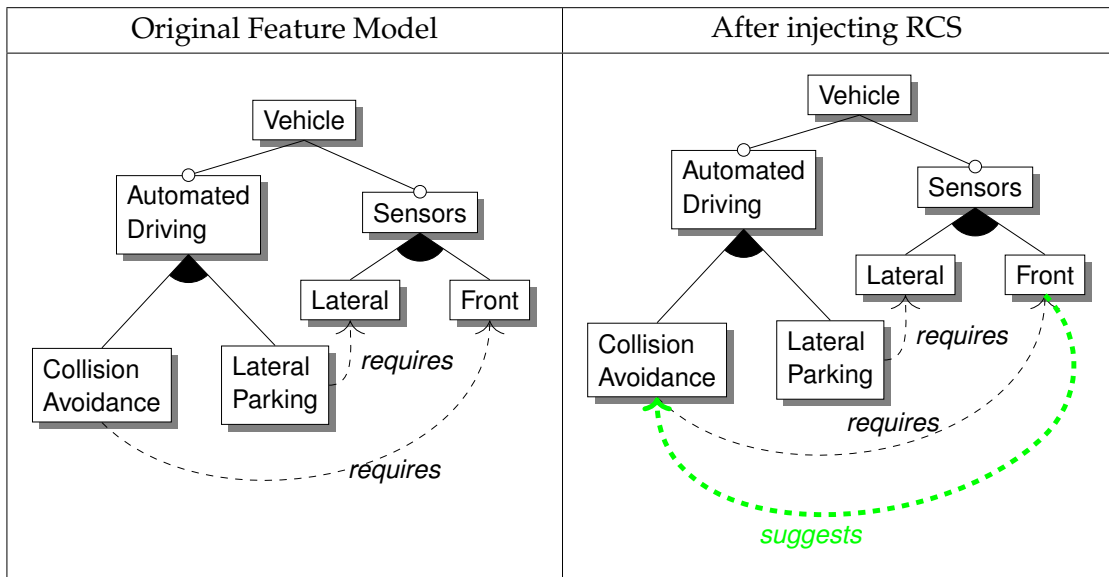


Figure 4.3: Example of Reverse Constraint Suggestion

#### 4.2.4 Group Selection Suggestion

**Overview:** The pattern Group Selection Suggestion (GSS) is related to the preferential configuration options of grouped features.

**Context:** A feature model includes a group  $G$  with subfeatures  $G_1, G_2, \dots$ . Preferred choices for the subfeatures can be identified by the domain engineer.

**Problem:** In standard feature models, there is no specific provision for describing preferred alternatives. This makes it impossible to record, reason or advise the user according to those preferences. If the application engineer is oblivious to such preferences, suboptimal configurations may be created.

**Solution:** A GSS is included in the feature model to describe preferred group options. A GSS is a soft constraint that describes preferred group configurations of group  $G$  (e.g.,  $GSS(G) = G \text{ suggests } G_x$ ).

**Example:** Figure 4.4 shows an example of injection of a GSS into a feature model. In this case, selection of the most advanced automated driving feature is recommended. More complex examples could include preferences contingent on the selection status of other features (e.g.,  $A \text{ and } G \text{ suggests } G_1$ ) or multiple alternative preferences (e.g.,  $G \text{ suggests } G_1 \text{ or } G_2$ )

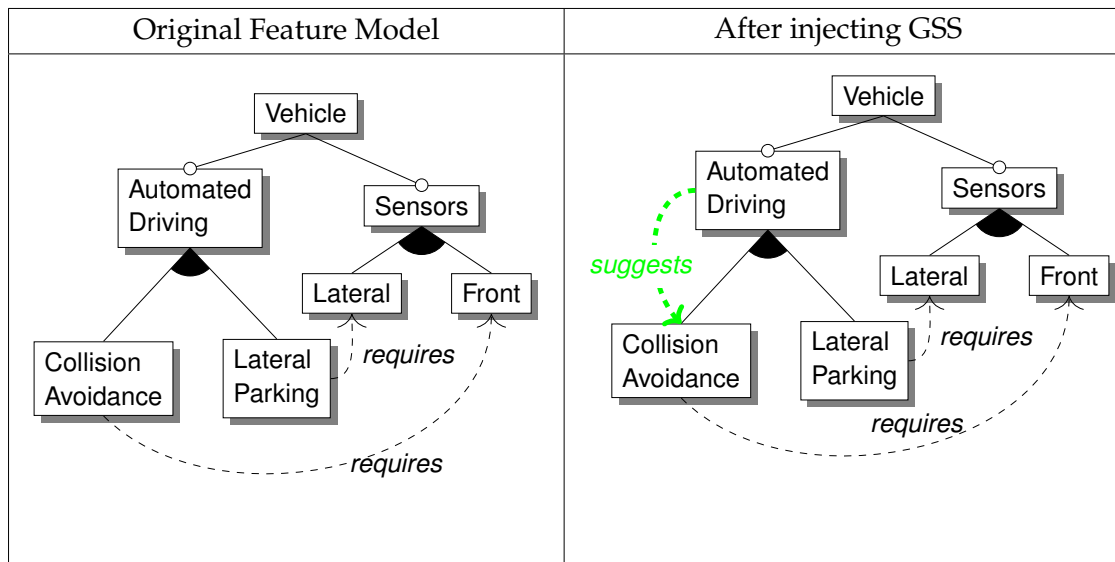


Figure 4.4: Example of Group Selection Suggestion

### 4.2.5 Soft Constraints and Feature Model Evolution

Another role of soft constraints can play in Domain Engineering is as a mechanism for representing the evolution of feature models. A generalization of a feature model is a transformation that increases the number of valid products [TBK09]. If a hard constraint is transformed into an equivalent soft constraint, a generalization may be performed less abruptly than just plain removal<sup>1</sup>. This is

<sup>1</sup>Strictly speaking, simply replacing a hard constraint by its soft counterpart does not necessarily imply that the number of products increases. This happens, for example, if the original

relevant if the original constraint contains meaningful domain information that applies to most products, but prevents a few specific products of interest to be created. Outright removal of the constraint, in such a situation, can be harmful as the implicit guidance it provides during the configuration process is no longer available. Another way of looking at the same situation is simply that the information represented by the original hard constraint becomes, for some external reason, better represented as a soft constraint.

Figure 4.5 presents an example where such a transformation might be applied. In the original feature model, the "Collision Avoidance" feature requires both lateral and front sensors, for improved performance. However, a front sensor is found to be sufficient for offering useful service, and it is determined that products with that configuration should be allowable, even though the original configuration is still recommended. In this way, the hard constraint is relaxed and now, even though the lateral sensor is no longer required, it is still suggested to maximize collision avoidance performance.

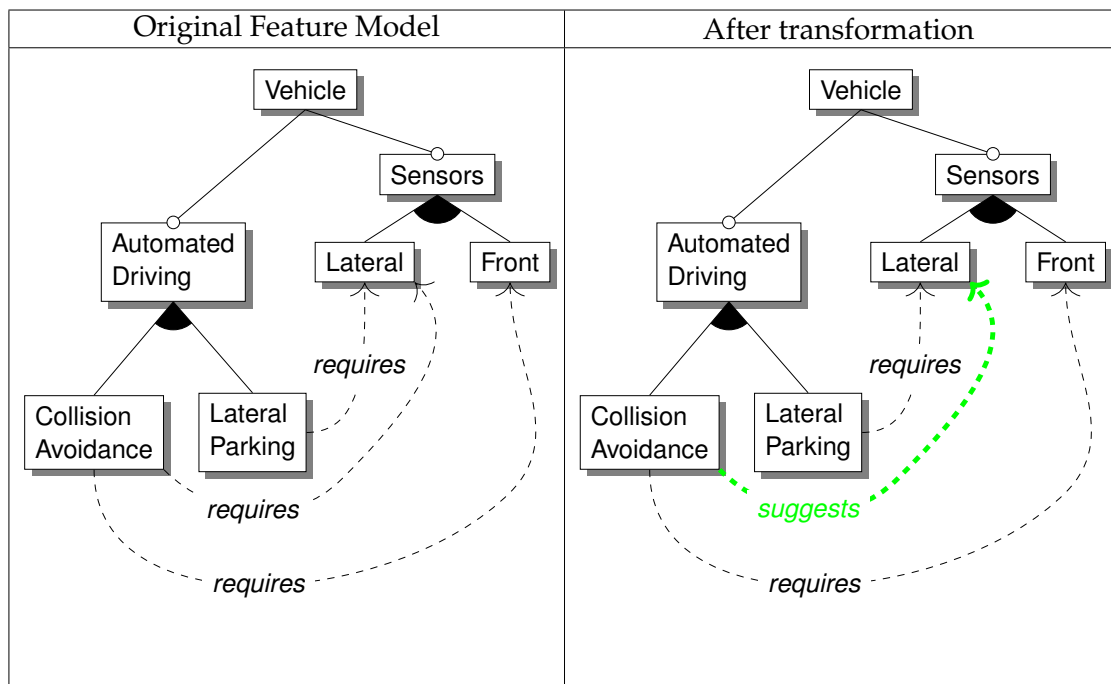


Figure 4.5: Evolving a feature model via soft constraints

The same concept can be applied to restructuring operations of the feature tree. In this case, the underlying constraints associated with the feature tree can be relaxed and turned into soft constraints. Figure 4.6 represents two examples

constraint is redundant (might be removed without an actual impact in product generation). In this case, the transformation would be better classified as a refactoring. The exact nature or classification of the transformation is not relevant for our discussion.

of this application.


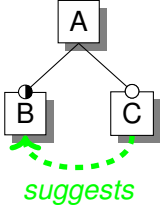

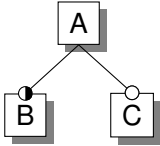
Original Feature Tree	Transformation	Resulting Feature Tree
 <p><i>B is mandatory or optional subfeature of A.</i></p>	Relaxing optional feature <i>C</i>	
 <p><i>B is mandatory or optional subfeature of A.</i></p>	Relaxing mandatory feature <i>C</i>	 <p><i>A suggests <math>B \Leftrightarrow C</math></i></p>

Figure 4.6: Restructuring the feature tree via soft constraints

### 4.3 Suspicious Soft Constraint Interactions

Even though soft constraints, by definition, can be disregarded when considering the validity of a configuration, they represent relevant domain information that can be automatically processed to provide useful feedback to the Domain Engineer, when he annotates the feature model with soft constraints. Of special interest to the domain engineer are the interactions of the soft constraints (both with other soft constraints and generally with the entirety of feature model) that can plausibly be considered suspicious. It turns out that these interactions can take different forms and colloquial designations such as "inconsistent", "invalid", "conflicting" or "interfering" may prove vacuous if an exact meaning is not established in this context. Therefore, we seek to determine and categorize what types of interactions that may prove useful for the domain engineer to learn about. These interactions are considered suspect because their purposeful and deliberated use can be considered unlikely, making it advisable to alert the domain engineer to their presence, so that human modeling error or unexpected results can be averted or rectified if necessary.

### 4.3.1 Suspicious Interaction Classification

Perhaps the most straightforward scenario that can be classified as suspicious corresponds to soft constraints that are not satisfied in any valid configuration, due to the interaction of the constraint expression with the feature tree structure and hard constraints. We designate these types of constraints as *unsatisfiable*. For example, the soft constraint "A *discourages* B" in Figure 4.7 is unsatisfiable.

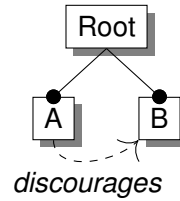


Figure 4.7: Unsatisfiable soft constraint example

Unsatisfiable soft constraints certainly do not invalidate any model. However, considering that a soft constraint may correspond to a desired (albeit optional) property of the domain, unsatisfiability is a testament to the impossibility of achieving it with the current model. In this way, the domain engineer should have a keen interest in being informed about such scenarios, so that he may assess if some kind of revision or corrective action is required.

Nevertheless, unsatisfiable soft constraints are not the only case of interest warranting automated detection and reporting. Some other scenarios, such as the ones illustrated in Figure 4.8 and Figure 4.9, can be considered quite possibly erroneous or at least very suspicious from a domain analysis perspective. However, it is worth noting that none of the constraints in this scenario are in fact unsatisfiable (neither individually nor even when grouped). Therefore, a different categorization is required for these scenarios.

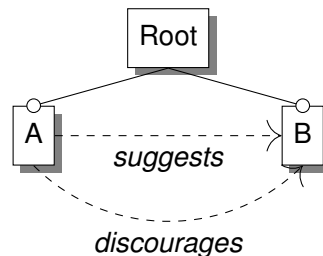


Figure 4.8: Contradictory soft constraints

In Figure 4.8, both constraints can be simultaneously satisfied by any configuration where feature *A* is not selected. However, these constraints seem to be providing contrary advice. We designate such constraints as *contradictory*, as

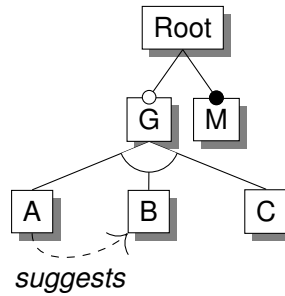


Figure 4.9: Untriggerable soft constraint Example

these cannot be satisfied if simultaneously triggered (that is, their implicant is true). Specifically, a set of soft constraints is contradictory if simultaneous triggering is possible, but simultaneous satisfaction is not. Contradictory constraints are not necessarily inconsistent, as the example of Figure 4.8 demonstrates, because simultaneous triggering may be avoided. However, the combined effect of these constraints is not clearly conveyed by their original formulation. Moreover, contradictory soft constraints will allow valid (partial) configurations that cannot be completed successfully while satisfying all soft constraints, without making retroactive changes. For example, it is not possible to obtain, without backtracking, a valid configuration satisfying all the soft constraints from the partial configuration where features *Root* and *A* are selected and feature *B* is still open (undecided). This may be frustrating for the users who will later attempt to create configurations simultaneously satisfying those constraints. Of course, it is up to the domain engineer to decide whether or not the obscure formulation or later configuration issues are a concern. In this case, automated analysis identifies and points out contradictory constraints to the domain engineer, who then has the responsibility of deciding whether or not any corrective action is required.

Figure 4.9 presents another scenario where the formulation of the soft constraint can be deemed suspicious. In this case, the soft constraint seems to be suggesting the concomitant selection of alternative features, which is obviously impossible. This is a specific case of the scenario where the implicant and implicated portions of the soft constraint cannot be satisfied simultaneously. Again, however, the soft constraint is not inconsistent: it is in fact satisfied in all valid configurations that do not trigger it, that is, where feature *A* is deselected. We designate soft constraints that can only be satisfied in this way as *untriggerable*. Specifically, a soft constraint is untriggerable if its implicant can be satisfied, but the constraint is satisfiable in its entirety only if untriggered. Much like contradictory constraints, untriggerable constraints suffer from very much the same obscure formulation problem: the soft constraint in Figure 4.9 can be replaced by an

equivalent and presumably clearer representation of its effect as a suggestion to unconditionally deselect  $A$ . However, while contradictions always involve two or more constraints, untriggerable soft constraints can occur independently.

It is worth pointing out that, although the interactions in Figures 4.7, 4.8, and 4.9 are fairly simply to detect by casual visual inspection, similar interactions may have a more obscure presentation in more complex feature models, making the use of automated detection mechanisms desirable.

To summarize, the following can be considered interactions of interest that warrant detection and identification:

- **Unsatisfiable soft constraints.** These cannot be satisfied in any valid configuration.
- **Contradictory soft constraints.** These cannot be triggered simultaneously in any valid configuration.
- **Untriggerable soft constraints.** These cannot be triggered in any valid configuration.

### 4.3.2 Identification of Suspicious Interactions

In this section, we demonstrate how the suspicious interactions described in the Section 4.3 can be detected. These algorithms can be used in a feature model editing tool, to provide immediate feedback to the user as soft constraints are added to a model. The purpose is only to identify the interactions: it is up to the Domain Engineer to decide whether or not some kind of corrective action is required.

#### 4.3.2.1 Identifying Unsatisfiable Soft Constraints

Identifying unsatisfiable soft constraints is akin to consistency analysis in standard feature modeling<sup>2</sup>, in the sense that if the unsatisfiable soft constraint was to be considered a hard constraint instead, then the resulting feature model would be inconsistent (i.e., no valid configuration would exist). Accordingly, standard consistency analysis approaches (e.g., SAT analysis) can be used to identify unsatisfiable soft constraints. Specifically, given the valid feature model expression  $F$  and soft constraint  $C$  with expression  $S_C$ ,  $C$  is unsatisfiable if:

$$\neg \text{SAT}(F \wedge S_C) \tag{4.1}$$

---

<sup>2</sup>We use the expression "standard feature modeling" to designate feature modeling without recourse to soft constraints

For an example of application, let's consider the feature model of Figure 4.7. Applying the transformations described in Section 2.2.1 to obtain the feature model expression  $F$ , we find that

$$\begin{aligned} F &= (A \Leftrightarrow \top) \wedge (B \Leftrightarrow \top) \\ S_C &= (A \Rightarrow \neg B) \end{aligned}$$

and applying (4.1),

$$\begin{aligned} \neg \text{SAT}(F \wedge S_C) &= \neg \text{SAT}((A \Leftrightarrow \top) \wedge (B \Leftrightarrow \top) \wedge (A \Rightarrow \neg B)) \\ &= \neg \text{SAT}(A \wedge B \wedge (A \Rightarrow \neg B)) \\ &= \neg \text{SAT}(A \wedge B \wedge (\neg A \vee \neg B)) \\ &= \neg \text{SAT}(\perp) \\ &= \top, \end{aligned}$$

we find the constraint "A" *discourages* "B" to be unsatisfiable.

#### 4.3.2.2 Identifying Contradictory Soft Constraints

A set of soft constraints is contradictory if simultaneous triggering is possible, but simultaneous satisfaction is not. Therefore, given the feature model expression  $F$ , a set of  $n$  soft constraints  $C_i$  with  $i = 1 \dots n$ , then the set is contradictory if:

$$\text{SAT}(F \wedge \bigwedge_{i=1}^n T(C_i)) \wedge \neg \text{SAT}(F \wedge \bigwedge_{i=1}^n (T(C_i) \wedge S_{C_i})) \quad (4.2)$$

where  $T(C_i)$  and  $S_{C_i}$  are the trigger and expression of  $C_i$ , respectively. This expression detects a contradiction among exactly  $n$  constraints. Other scenarios must be addressed separately.

Considering the example of Figure 4.8,

$$\begin{aligned} F &= (A \Rightarrow \top) \wedge (B \Rightarrow \top) \\ &= (\neg A \vee \top) \wedge (\neg B \vee \top) \\ &= \top \\ S_{C_0} &= (A \Rightarrow \neg B) \\ S_{C_1} &= (A \Rightarrow B) \\ T(C_0) &= A \\ T(C_1) &= A \end{aligned}$$

we find that,

$$\begin{aligned}
 \text{SAT}(F \wedge \bigwedge_{i=1}^n T(C_i)) &= \text{SAT}(\top \wedge \bigwedge_{i=1}^2 T(C_i)) \\
 &= \text{SAT}(T(C_0) \wedge T(C_1)) \\
 &= \text{SAT}(A \wedge A) \\
 &= \text{SAT}(A) \\
 &= \top
 \end{aligned} \tag{4.3}$$

and also

$$\begin{aligned}
 \neg \text{SAT}(F \wedge \bigwedge_{i=1}^n (T(C_i) \wedge S_{C_i})) &= \neg \text{SAT}(\top \wedge \bigwedge_{i=1}^2 (T(C_i) \wedge S_{C_i})) \\
 &= \neg \text{SAT}((T(C_0) \wedge S_{C_0}) \wedge (T(C_1) \wedge S_{C_1})) \\
 &= \neg \text{SAT}((A \wedge (A \Rightarrow \neg B)) \wedge (A \wedge (A \Rightarrow B))) \\
 &= \neg \text{SAT}((A \wedge (\neg A \vee \neg B)) \wedge (A \wedge (\neg A \vee B))) \\
 &= \neg \text{SAT}((A \wedge \neg B) \wedge (A \wedge B)) \\
 &= \neg \text{SAT}(\perp) \\
 &= \top
 \end{aligned} \tag{4.4}$$

Replacing (4.3) and (4.4) into (4.2), we find the soft constraints  $S_0$  and  $S_1$  to be contradictory in the feature model of Figure 4.8.

### 4.3.2.3 Identifying Untriggerable Soft Constraints

A soft constraint is untriggerable if it can be satisfied only if not triggered. In this way, a constraint  $C$  with expression  $S_{C_i}$  and trigger  $T(C)$  is untriggerable if

$$\text{SAT}(F \wedge S_C) \wedge \neg \text{SAT}(F \wedge S_C \wedge T(C)) \tag{4.5}$$

Applying (4.5) to the example in Figure 4.9, we find that

$$\begin{aligned}
 F &= A \Leftrightarrow (G \wedge \neg B \wedge \neg C) \wedge \\
 B &\Leftrightarrow (G \wedge \neg A \wedge \neg C) \wedge \\
 C &\Leftrightarrow (G \wedge \neg A \wedge \neg B) \wedge \\
 (M &\Leftrightarrow \top) \wedge (G \Rightarrow \top) \\
 &= \dots
 \end{aligned}$$

$$\begin{aligned}
&= (\neg A \wedge \neg B \wedge \neg C \wedge \neg G \wedge M) \vee \\
&\quad (\neg A \wedge \neg B \wedge C \wedge G \wedge M) \vee \\
&\quad (\neg A \wedge B \wedge \neg C \wedge G \wedge M) \vee \\
&\quad (A \wedge \neg B \wedge \neg C \wedge G \wedge M) \\
&S_C = (A \Rightarrow B) \\
&T(C) = A.
\end{aligned} \tag{4.6}$$

Knowing that

$$\text{SAT}(f(x, y, \dots)) \Rightarrow \text{SAT}(f(x, y, \dots) \vee g(x, y, \dots)) \tag{4.7}$$

for any  $f(x, y, \dots)$  and  $g(x, y, \dots)$ , and also that

$$\text{SAT} \bigwedge_i x_i = \top, \tag{4.8}$$

provided that  $x_i \neq \neg x_j$ , for all  $i, j$ , then, considering from (4.6) that  $F = \bigvee_{i=0}^3 P_i$ , where

$$\begin{aligned}
P_0 &= (\neg A \wedge \neg B \wedge \neg C \wedge \neg G \wedge M) \\
P_1 &= (\neg A \wedge \neg B \wedge C \wedge G \wedge M) \\
P_2 &= (\neg A \wedge B \wedge \neg C \wedge G \wedge M) \\
P_3 &= (A \wedge \neg B \wedge \neg C \wedge G \wedge M),
\end{aligned}$$

then

$$\begin{aligned}
\text{SAT}(F \wedge S_C) &= \text{SAT}(F \wedge (A \Rightarrow B)) \\
&= \text{SAT}(F \wedge \neg A \vee F \wedge B) \\
&= \text{SAT}\left(\bigvee_{i=0}^3 (P_i \wedge \neg A) \vee \bigvee_{i=0}^3 (P_i \wedge B)\right)
\end{aligned} \tag{4.9}$$

By (4.7) and (4.9), it is enough to demonstrate  $\text{SAT}(P_i \wedge \neg A)$  or  $\text{SAT}(P_i \wedge B)$ , for some  $i$ , to demonstrate  $\text{SAT}(F \wedge S_C)$ . Making  $i = 0$ , we find that:

$$\begin{aligned}
\text{SAT}(P_0 \wedge \neg A) &= \text{SAT}((\neg A \wedge \neg B \wedge \neg C \wedge \neg G \wedge M) \wedge \neg A) \\
&= \text{SAT}(\neg A \wedge \neg B \wedge \neg C \wedge \neg G \wedge M) \\
&= \top \text{ by (4.8)}
\end{aligned} \tag{4.10}$$

so by (4.7), (4.9) and (4.10), we conclude that

$$\text{SAT}(F \wedge (A \Rightarrow B)) = \top \quad (4.11)$$

Also,

$$\begin{aligned} \neg \text{SAT}(F \wedge S_C \wedge T(C)) &= \neg \text{SAT}(F \wedge (A \Rightarrow B) \wedge A) \\ &= \neg \text{SAT}(F \wedge (\neg A \vee B) \wedge A) \\ &= \neg \text{SAT}(F \wedge B \wedge A) \\ &= \neg \text{SAT}\left(\bigvee_{i=0}^3 (P_i \wedge B \wedge A)\right) \\ &= \neg \text{SAT}\left((P_0 \wedge B \wedge A) \vee (P_1 \wedge B \wedge A) \right. \\ &\quad \left. \vee (P_2 \wedge B \wedge A) \vee (P_3 \wedge B \wedge A)\right) \\ &= \neg \text{SAT}(\perp \vee \perp \vee \perp \vee \perp) \\ &= \top \end{aligned} \quad (4.12)$$

Replacing (4.11) and (4.12) in (4.5), we find that  $S_C$  is correctly identified as untriggerable.

## 4.4 Conclusions

Soft constraints offer increased modeling opportunities in domain engineering. The information they convey cannot be comfortably represented by hard constraints, so, in the absence of soft constraints, it must be either left out or be misrepresented. This has unfortunate consequences, as the missing domain information could be put into good use by serving as the basis for analysis and reasoning about the properties of the feature model. It also could be used for providing better support during the configuration process and validation of the created configurations.

We have described several prototypical applications of soft constraints, by identifying their context of application, problem being addressed and solution. These applications relate preferential configurations options, suggest the efficient use of available resources, and identify preferred options in feature groups. We also illustrated the potential of soft constraints for feature model evolution. By relaxing a hard constraint into a corresponding soft constraint, it becomes possible to evolve the product line scope to include more product variants, without completely throwing away the information stored in a hard constraint that might

yet be somehow relevant.

We also present a categorization of suspicious soft constraint interactions. These suspicious interactions represent scenarios that are unlikely to have been deliberately created by the domain engineer, although no actual error is present. We also present the mechanism for performing the automated identification of these interactions, so that they may be reported to domain engineer, who then must assess the situation to determine if any kind of corrective measure is required or not. This showcases the usefulness of soft constraints, as these validations are only possible due to their inclusion in the feature model.





# Enhanced Configuration Support

Standard feature model configurators are primarily concerned with ensuring that the user always achieves a valid configuration. This is accomplished by propagating user choices as required, in an iterative process where a partial configuration incrementally grows from empty up to complete. Although this ensures that a valid solution is always obtained, it does not necessarily help the stakeholder to achieve his vision. We propose enhanced support for the configuration process (Section 5.1), by providing algorithms for computing configuration suggestions, and conflict identification and explanation (Section 5.2), as well as a prototype tool we used for test and validation purposes (Section 5.3). This chapter provides material addressing research questions 1 (by clarifying the role of soft constraints with respect to the goal of our work), 1b (by explaining the role of soft constraints in the configuration step), 2 (by describing enhanced configuration support that provides configuration suggestions and trade-off analysis), and 3a (by presenting an algorithm for detecting and explaining conflicted features in overconstrained configurations).

## 5.1 Enhanced Support Overview

Our enhanced support approach [BM13, BM14b] is based on two key functionalities:

- **Providing Configuration Suggestions**, which are generated dynamically

during an interactive configuration process.

- **Identifying and explaining conflicts**, which are scenarios where the simultaneous satisfaction of all applicable soft constraints is not possible.

We describe these in greater detail in Sections 5.1.1 and 5.1.2, respectively.

### 5.1.1 Configuration Suggestions

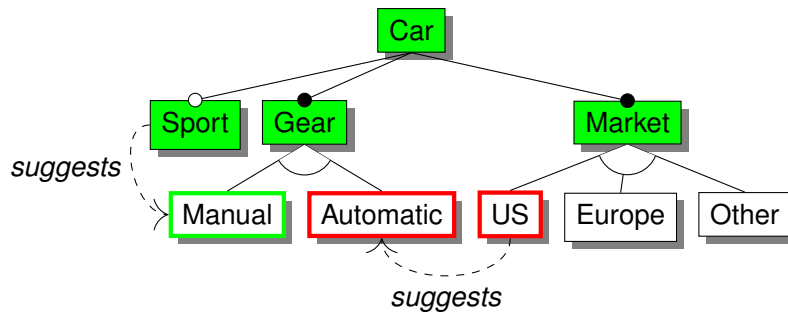
Suggestions are provided during the iterative configuration of the feature model. They are based on available soft constraints, which can represent either user preferences or domain information. The stakeholder may decide to follow this advice, but is free to ignore it. Nevertheless, inspection of the advice gives the stakeholder the opportunity to revise his original intentions, if necessary, or it might help him to decide upon a specific configuration aspect he was unsure of. Regardless, if he chooses to do so, he may proceed against the provided advice, but does so knowingly. Figure 5.1a presents an example of the intended advise behavior. Considering the partial configuration, the system can advise the user to select the "Manual" transmission feature, and deselect the "Automatic" and "US" features. This advice takes into consideration the current partial configuration. The "Sports" feature is already selected, so it is necessary to select "Manual" to ensure that the "Sports" suggests "Manual". On the other hand, selecting the "Manual" feature entails deselection of the "Automatic" feature (and vice-versa), so "Automatic" should be deselected. Finally, if "Automatic" is deselected, it is necessary to deselect "US" to satisfy the soft constraint "US" *suggests* "Automatic". Therefore, the advice generated by the system ensures that, if followed, both constraints can be satisfied. However, this advice is not mandatory: the user may deviate from the suggestions provided by the configurator. However, in that case, it will not be possible to uphold all constraints. Figure 5.1b provides an example of that situation: the user has determined not to follow the advice and continues the example of Figure 5.1a by selecting the "US" feature. At this point, it is still possible to satisfy either one of the constraints, but not both. Which one gets satisfied depends on future configuration actions. Selection of either the "Automatic" or "Manual" feature will lead to different outcomes. It should be pointed out that, although it is obvious that an advice to deselect "USA" in Figure 5.1b would suitably point towards maximal constraint satisfaction, it would simply restate the previous advice given in Figure 5.1a, which has already been directly ignored by the user. To avoid such situations, and to provide useful assistance no matter what choices are made, this approach provides advice only for open features. The configurator

will never advise any change to the state of a feature that is already selected or deselected by virtue of user action.

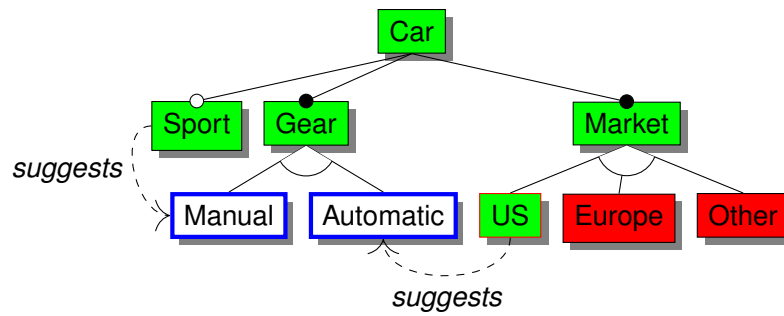
### 5.1.2 Conflict Identification and Explanation

Rather than focusing on the validity of the configuration, which is handled implicitly by the iterative configuration algorithm, we just intend to identify and report potential conflicts during the configuration process, in terms of the soft constraint satisfaction. Specifically, we seek to identify and report to the stakeholder *conflicted features*. A conflicted feature is an open feature in a partial configuration whose selection or deselection will necessarily entail not satisfying some (different) soft constraints. For example, in Figure 5.1b, both "Automatic" and "Sports" are conflicted features. The information provided below the feature diagram explains the reasons supporting either the selection or deselection of each conflicted feature. A stakeholder may analyze such information before deciding what is the best way of proceeding with the configuration process. Selection of "Automatic" is necessary (and sufficient) for satisfying the "US" *suggests* "Automatic" constraint, but prevents satisfaction of the other soft constraint. The converse situation is found when selecting "Manual". Conflicted features may arise not only due to overconstraining the model, but also by the user making choices that disregard the impact of soft constraints. This is always a possibility that must be considered, since soft constraint satisfaction is not mandatory. Conflicted features represent required trade-offs that are to be decided by the stakeholder. By choosing to either select or deselect a conflicted feature, the stakeholder is deciding which constraint or constraints he deems more relevant to satisfy, in detriment of other(s). The use of a priority-based scheme to categorize the relative importance of the SCs could be considered, with the idea of allowing the automatic resolution of conflicts in favour of the higher priority SCs without requiring stakeholder intervention. However, obtaining the total ordering of SCs required to achieve this is not simple, requiring great consideration and care. Some of this effort will be wasted because some constraints may not conflict at all. Also, priorities are not necessarily static and can depend not only on the stakeholder but also on the specific configuration that is being considered. By deferring resolution of these conflicts to the configuration (realization) step, we allow a specific context (product) to be considered by the stakeholder, so that a decision can be made based on concrete factors rather than abstract *a priori* considerations.

It is possible that, in some cases, multiple SCs are involved in a conflict. To address these scenarios, a metric can be calculated that provides some direction.



(a) Configuration suggestions



Manual	want to select due to "Sports" suggests "Manual"
Automatic	want to deselect due to "US" suggests "Automatic"
	want to select due to "US" suggests "Automatic"
	want to deselect due to "Sports" suggests "Manual"

(b) Conflicted Features

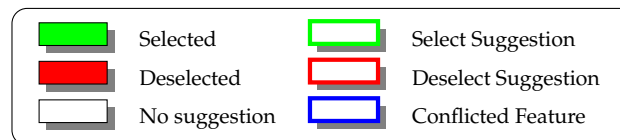


Figure 5.1: Enhanced configuration support example

Consider the example of Figure 5.2, where an additional feature and soft constraint has been added to the vehicle feature model presented in Figure 5.1. In this case, there are two SCs whose satisfaction is dependent on selection of the "Manual" feature, while a single one depends on deselection of that feature. All else being considered equal and looking at the raw number of satisfied SCs, selection of the "Manual" feature seems to be the preferable option. If the satisfaction of even more SCs was dependent on selection of "Manual", that recommendation could be considered even stronger. Accordingly, this type of information can be provided to the user in the form of a metric in the range -1 to +1, where +1 corresponds to an absolute recommendation of selection and -1 an absolute recommendation to deselect, and intermediate values recommendations of milder strength. For this purpose, given the number  $N_{s,f}$  of SCs that require selection of a feature  $f$  and the number of SCs requiring its deselection ( $N_{d,f}$ ), we compute the metric  $R_f$ :

$$R_f = \frac{N_{s,f} - N_{d,f}}{N_{s,f} + N_{d,f}} \quad (5.1)$$

as an auxiliary feedback to the user. In the case of a conflict caused by a balanced number of SCs, such as the one in in Figure 5.1b,  $N_{s,f} = N_{d,f}$  and  $R_{Automatic} = R_{Manual} = 0$ . No clear indication for selection or deselection is discernible. For the example in Figure 5.2,  $R_{Automatic} = -\frac{1}{3}$  and  $R_{Manual} = \frac{1}{3}$ , corresponding to a mild strength suggestions to deselect "Automatic" and select "Manual", respectively. Nevertheless, the resulting value is only a synthesis of the complete information provided, and can be easily replaced by other metric or disregarded if necessary.

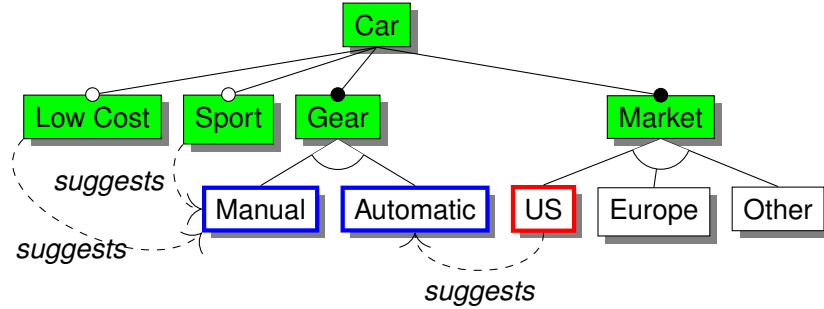


Figure 5.2: Features conflicted due to multiple constraints

## 5.2 Algorithms for Configuration Advice and Conflict Analysis

To determine configuration advice as presented in Figure 5.1a, we resort to Algorithm 1. The algorithm begins (line 1) by identifying the *active* SCs. An active

---

### Algorithm 1: Suggestion computation

---

**input** : Partial configuration config, feature model fm and constraints  
**output**: Configuration suggestions

```

1 active ← activeSoftConstraints (config, fm)
2 if (sat (fm · config · active)) then
3   selSuggestions ← computeCommonFeatures(fm, active · config)
   /* selSuggestions now holds features that must be selected to
   ensure satisfaction of all soft constraints. */
4   deselSuggestions ← computeDeadFeatures(fm, active · config)
   // eval maps features to a (de)selection score
5   foreach f in selSuggestions do
6     eval [f] ← 1
7   foreach f in deselSuggestions do
8     eval [f] ← -1
9   return eval
10 else
11   return analyzeInconsistency ()

```

---

soft constraint is one that may still be satisfied in the partial configuration. Once the user makes choices that render satisfaction of a SC impossible, that constraint is in fact disregarded. No advice is ever provided to attempt the retroactive satisfaction of the constraint after choices have been made that prevent its satisfaction, as discussed in the previous section.

Algorithm 2 is used to compute the active soft constraints. It returns a conjunction of the expressions that satisfy all the active soft constraints. This is achieved by verifying the satisfiability of each soft constraint in the current configuration (Algorithm 2, line 4).

After determining the active constraints, Algorithm 1 proceeds by verifying the satisfiability of all active SCs in the current partial configuration (line 2). If it is satisfiable, then it is possible to complete the configuration while satisfying all SCs. In this case, the configurator computes common and dead features as if the SCs were hard constraints (lines 3 and 4). The first must be selected so that all SCs are satisfied, while the latter must be deselected. In this way, dead

**Algorithm 2:** Active feature computation

---

```

1 Function activeSoftConstraints
  input : Partial configuration config, feature model fm and constraints
  output: Conjoined expressions of active soft constraints

2   active  $\leftarrow$  true
3   foreach soft constraint  $S$  do
4     if (sat ( $S \cdot \text{config} \cdot \text{fm}$ )) then
5       active  $\leftarrow$  active  $\cdot S$ 
6   return active

```

---

features correspond to an advice to deselect, while common features result in advices to select. Advices to select are represented by associating a score of +1 to features that should be selected, and  $-1$  to features that should be deselected (lines 6 and 8). If, however, it is not possible to satisfy all SCs in the current partial configuration, then a conflict exists. Line 11 calls Algorithm 3, which identifies the conflict and collects the relevant information.

When Algorithm 3 is run, it has already been established that it is not possible to satisfy simultaneously all active SCs (line 4 of Algorithm 1). Therefore, the task is now to identify and provide a valid explanation for this impossibility. The likelier scenario is that it is possible to satisfy *some* of the active SCs, or maybe even most, but not all. Having this in mind, the algorithm will begin its analysis by iterating over all  $k$ -combinations of SCs, beginning with  $k = 1$  up to  $N$ , where  $N$  is the number of active constraints (line 5). The algorithm will always terminate at most when  $k = N$ , since it has already been established that the active SCs are not simultaneously satisfiable. As soon as an explanation for the conflict is found (line 17), the iteration can stop, so it is not necessary to fully iterate over the entire range of  $k$ .

To provide the necessary feedback to the user, the conflicted features must be identified. This is achieved by identifying all features that must be dead or common to ensure satisfiability of each  $k$ -combination of constraints. A feature is identified as conflicted if it must be selected (common) to satisfy some  $k$ -permutation of SCs, while simultaneously being required its deselection (e.g., being found dead) to achieve satisfaction a different permutation (line 17). As soon as at least one conflicted feature is found, an explanation for the conflict has already been found. By iterating from  $k = 1$  upwards, the algorithm will identify the simplest explanations for the conflict first (by simplest, we mean those

**Algorithm 3:** Analyze inconsistency

---

```

1 Function analyzeInconsistency
   input : Partial configuration config, feature model fm and constraints
   output: Conflicted features, associated metrics and explanation.

2   done  $\leftarrow$  false
3   k  $\leftarrow$  1
4   while  $\neg$  done do
5     foreach k-combination S of active soft constraints do
6       dead  $\leftarrow$  computeDeadFeatures (fm · S, config)
7       common  $\leftarrow$  computeCommonFeatures (fm · S, config)
8       /* associate soft constraint satisfaction to required
9         (de)selection of features. */
10      if not empty dead then
11        foreach feature f in dead do
12          deselect [f]  $\leftarrow$  true
13          Add S to deselectionExplanation [f];
14      if not empty common then
15        foreach feature f in common do
16          select [f]  $\leftarrow$  true
17          Add S to selectionExplanation [f];
18      if  $\neg$ done then
19        done  $\leftarrow$  there exists f such that deselect [f] · select [f]
20      k  $\leftarrow$  k + 1
21  return computeMetric(deselectionExplanation, selectionExplanation)

```

---

involving the fewer number of constraints) and the corresponding conflicted features will also be identified. The algorithm also stores information that can be used to trace conflicted features to the relevant conflicting permutations of SCs (lines 11 and 15). This will allow feedback like the one illustrated in Figure 5.1b, where conflicted features are explained in terms of impact on SC satisfaction, to be produced. Finally, in line 19, (5.1) is used to compute a feedback metric based on the gathered information.

### 5.3 Tool Description

We developed a prototype tool that supports configuration of feature models annotated with soft constraints and provides enhanced support as described in this section. The layered structure of the tool is presented in Figure 5.3.

- **Presentation:** This package includes all the user-interface components. It is directly dependent on all domain logic.
- **Logic:** This package provides logic representation, evaluation and parsing capabilities for other modules. It relies on the Sat4j external library for resolving satisfiability problems.
- **FeatureModeling:** Provides feature model management capabilities. It relies on the **Logic** package for Boolean logic services, on the external SPLOT file reader library and the **SoftConstraint** package for representing soft constraints associated with the feature model.
- **SoftConstraint:** Provides soft constraint representation and evaluation. It relies on the **Logic** package for Boolean logic services and on the **FeatureModeling** package. The latter is required as evaluation of a soft constraint satisfaction must be considered in the context of a specific feature model.
- **Configuration:** Provides configuration support services. It depends on the **FeatureModelling** and **SoftConstraint** packages, as these provide the required context (i.e., the involved variables, feature model and hard and soft constraints) necessary for the configuration activity. It also relies on the **Logic** package for satisfiability services.
- **Project:** This package provides the required services for allowing management and persistence feature modeling, soft constraints and related configurations. Correspondingly, it is dependent on **FeatureModeling** and **Configuration**.

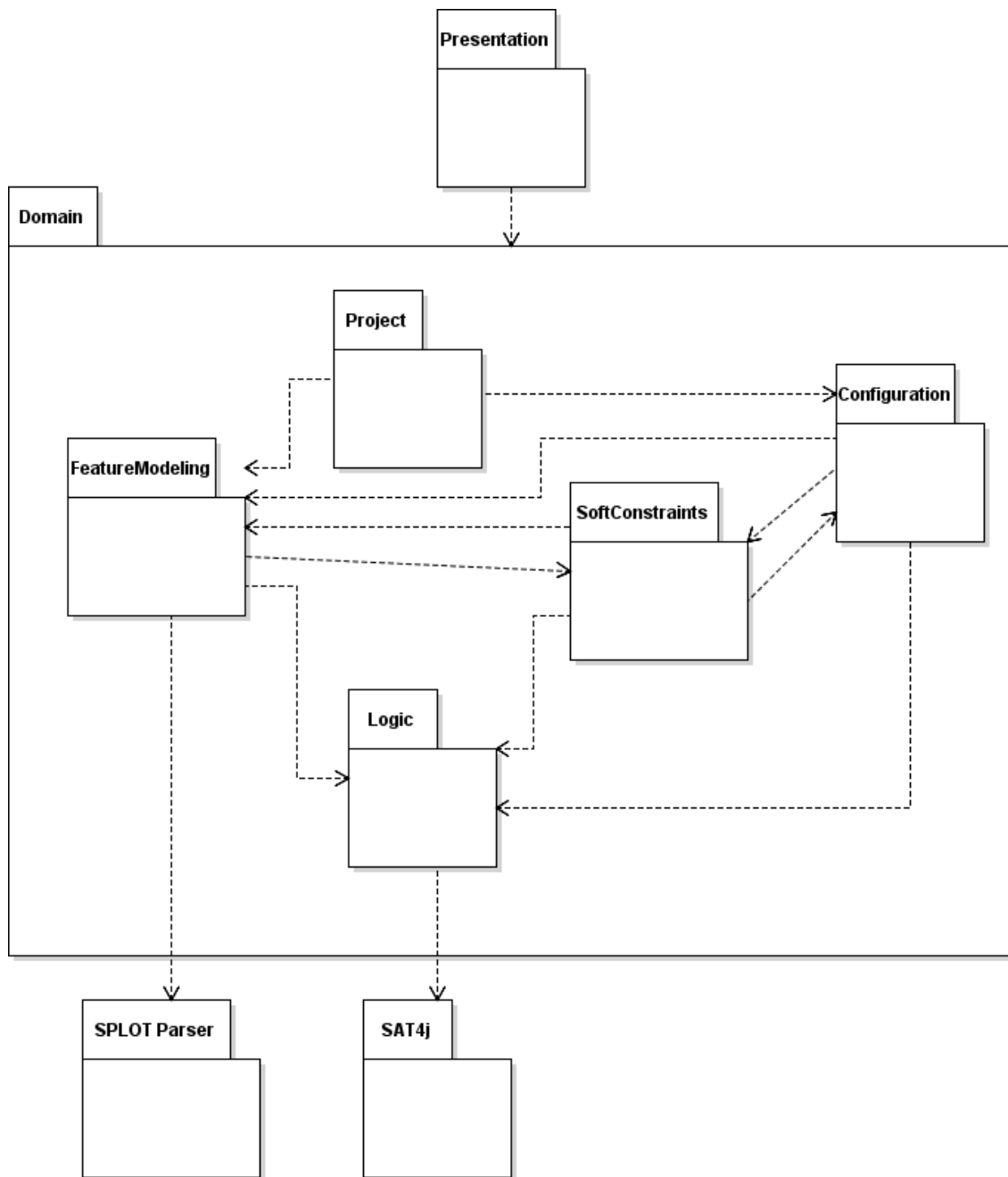


Figure 5.3: Configurator tool packages

Our tool was used for validation purposes, as described in Chapter 7. Figure 5.4 presents a screenshot at the start of the configuration process of one of the experiment test cases, where a portable computer is configured, according to user preferences represented as soft constraints. The feature tree can be seen in the left side of the screen ①, while on the right side, from top to bottom, we can find the hard constraints ②, soft constraints ③ and the explanation ⑤ for the currently highlighted conflicted feature ④. The soft constraints represent, in this case, desired properties of the configuration (e.g., it should include an optical drive, the optical drive should be a dvdrom and cdrw combo or a bluray drive, etc). None of these properties are mandatory: in fact, it turns out it is impossible to satisfy all those constraints simultaneously. The enhanced configuration support will aid the stakeholder in understanding the conflicted features and required trade offs so he can make the necessary decision.

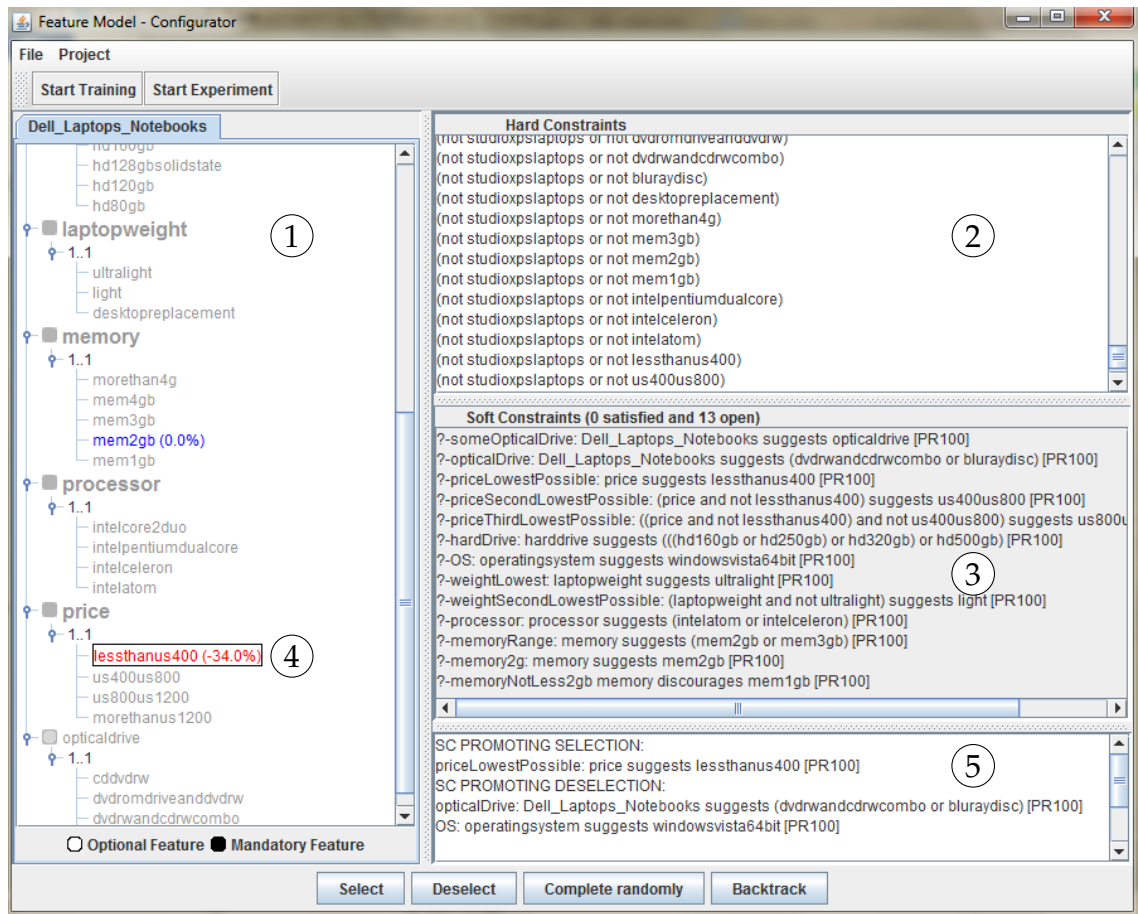


Figure 5.4: Configurator tool - Experiment test case

A conflicted feature ("lessThanUs400") is highlighted on the feature tree ④. The recommendation is to deselect with strength  $-34\%$ . The explanation can be

found in the bottom right panel (5). Selecting that feature is required to satisfy the soft constraint named "priceLowestPossible" ("price" suggests "lessThanUS400"). However, deselecting it is required to satisfy the "opticalDrive" soft constraint ("Dell\_Laptop\_Notebooks" suggests "dvdrwandcdrwcombo" or "bluray") and the "OS" soft constraint ("operatingsystem" suggests "windowsvista64bit"). While it should be fairly clear to the user that the "priceLowestPossible" soft constraint requires selection of "lessThanUS400", the impact on the "opticalDrive" and "OS" constraints is not so obvious since none of those constraints are directly involved with that feature. It would be necessary to acquire a deeper understanding of the feature model and hard domain constraints (over 100 in this case) to reach the same conclusions, if enhanced support was not provided. Another conflicted feature, "mem2gb", can also be observed higher up in the feature tree. In this case, no selection or deselection suggestion is provided (0%). Highlighting that feature would let the user know that selection of "memory2g" is required to satisfy the "memory2g" soft constraint ("memory" suggests "memory2g"), however deselecting it would be necessary to allow satisfaction of the "opticalDrive" constraint. Again we can see that the impact of selection or deselection of a conflicted feature is not always obvious or restricted to those constraints where it is directly referred to.

## 5.4 Conclusions

To achieve a more user-centric configuration approach, we propose to enhance the standard iterative configuration techniques so that configuration advice and conflict detection and analysis are provided to the stakeholder or domain engineer. Advice is provided to the user based on the soft constraint annotations and the current partial configuration. This advice allows the domain engineer to reconsider his planned course of action (that is, revising his idealized configuration) or finally deciding upon some configuration aspect that was still uncertain. Considering that soft constraints and the advice can be ignored by the application engineer, advice provided during the iterative process is concerned only with configuration possibilities still yet open. The configurator will defer to user choices and never advises undoing them, rather, it provides advice targeting only the part of the model which is still open (not configured).

Conflict detection is based on identification of conflicted features. These are features for which contradictory configuration options are required to satisfy some soft constraints. Algorithms were presented to detect these scenarios and

provide the required explanations. These explanations are provided in terms of the impact of each alternative choice in soft constraint satisfaction. The stakeholder can make an informed decision to resolve the conflict, having a clearer understanding of the consequences of selecting or deselecting the conflicted feature.

A prototype tool was developed, which implemented all the described algorithms and provided support for conducting the validation tests.



# 6

## Prototype-Based Configuration

Iterative configuration techniques require that the user specifies the configuration status of each feature in some arbitrary order of his desire. If a stakeholder uses this approach to specify a valid configuration in a standard iterative configurator (that is, the idealized configuration he is trying to create happens to be valid), then configuration order is not meaningful as the desired valid configuration will always be attained. However, if the idealized configuration is not valid, then it will not be achievable. The iterative configurator will propagate the configuration choices in such a way that valid completion is ensured. However, the exact configuration attained in this way is dependent on the order by which the features are specified. This can frustrate the user, as the trade-offs required to ensure validity are never made explicit. Rather, they are resolved implicitly according to the configuration order that happens to be adopted. In this chapter, we describe our *prototype-based* approach to configuration that seeks to address this problem (Section 6.1). At the core of this approach, a configuration repair technique (Section 6.2) is necessary for resolving configuration errors that may be introduced. Our cover-based configuration repair technique (Section 6.3) efficiently computes all possible fixes and allows for a concise representation of repair possibilities (Section 6.4). This algorithm has been implemented by our prototype tool (Section 6.5). Original contributions include the Prototype-based configuration approach, and a novel configuration repair algorithm. Prototype-based configuration allows the stakeholder to provide a complete description of

its idealized configuration to the system, so it is concerned with addressing research question 3.

## 6.1 Prototype-based vs. Iterative Configuration

The enhanced configuration support technique based in soft constraints, described on Chapter 5, helps the user to create a configuration meeting his wishes, that is, his idealized configuration. Nevertheless, it is still based on standard incremental configuration, so that the configuration is created step-by-step by selection/deselection of a single feature at a time. On the other hand, prototype-based configuration can be described as a generalization of an iterative approach to configuration, in which more than one feature can be selected or deselected simultaneously in each iteration. Ultimately, in the most extreme interpretation, the entire product can be specified in only a single iteration in which all features would be either selected or deselected as necessary. We will henceforth consider in the discussion that prototype-based configuration refers to this single iteration scenario, with the understanding that partial configuration with multiple iterations is also possible.

Prototype-based configuration is aligned with the idea of enabling the user to completely describe the idealized product in its entirety, unhindered by concerns of validity. This is often not the case when an iterative approach is taken, as choice propagation automatically configures certain features, depending on the partial configuration inputted so far, making it impossible for the stakeholder to fully configure the model according to his wishes. In other words, the idealized configuration may be invalid, but iterative configurators do not allow invalid (partial) configurations to be represented.

Figures 6.1 and 6.2 illustrate the order-dependency problem. Figure 6.1 represents a feature model and the idealized configuration of the stakeholder, which is invalid because the hard constraint "Custom" *requires* "Cabriolet" is not satisfied. Figure 6.2 represents the resulting outcome by trying to achieve the idealized configuration by specifying the features according to the idealized model, but in different orders. Figure 6.2a demonstrates the outcome obtained by selecting the "Gasoline" feature first, then the "Automatic" feature. At this point, due to automatic propagation of choices, the configuration becomes complete and the "Custom" feature configuration deviates from the idealized configuration. Another scenario is represented in Figure 6.2b, where configuration is made by selecting "Gasoline", then "Custom". Although the configuration is not yet complete, as

the user still must decide whether or not he prefers to have some kind of climate control (though his preferred option is no longer possible), a very different configuration is attained, also deviating from the ideal but in other ways. It is no surprise that the idealized configuration is not obtained in either case. This is a manifestation of its invalidity and some type of mismatch between the idealized and realized configurations is unavoidable. However, no help is provided to the user to help him understand what kind of trade-offs are required or possible: for example, what compromises would have to be made to allow for the selection of the "Custom" feature in the first scenario? Is some configuration order "better" than other? What trade-offs are involved? In a general case, answering questions like these requires either domain expertise (that the stakeholder might not possess nor be willing to acquire) or time-consuming backtracking and *ad-hoc* experimentation. Prototype configuration avoids, or at least mitigates this issue,

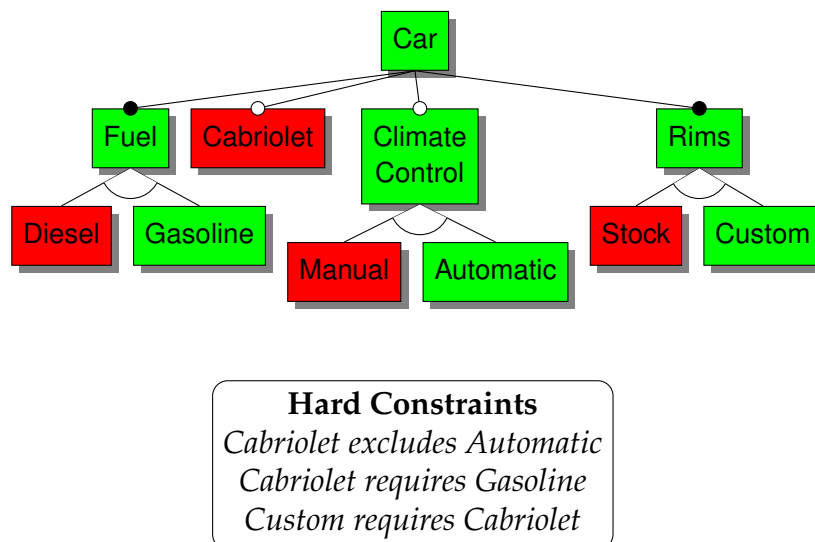


Figure 6.1: Feature model and idealized configuration

because it does not require that features are specified one by one. While the concept of prototype-based configuration is simple enough, one important hurdle is that when (de)selecting more than one feature simultaneously, it is possible that an invalid configuration is created. In fact, one of the reasons for the success of iterative solutions to configuration is that they completely avoid this problem. They achieve this by ensuring that the consequences of the (de)selection of a single feature are automatically applied as needed to ensure validity is preserved. This makes it impossible for the user to specify invalid configurations. Unfortunately, that is not the case if two or more features are simultaneously specified, as mutually exclusive options may be specified. This means that prototype-based

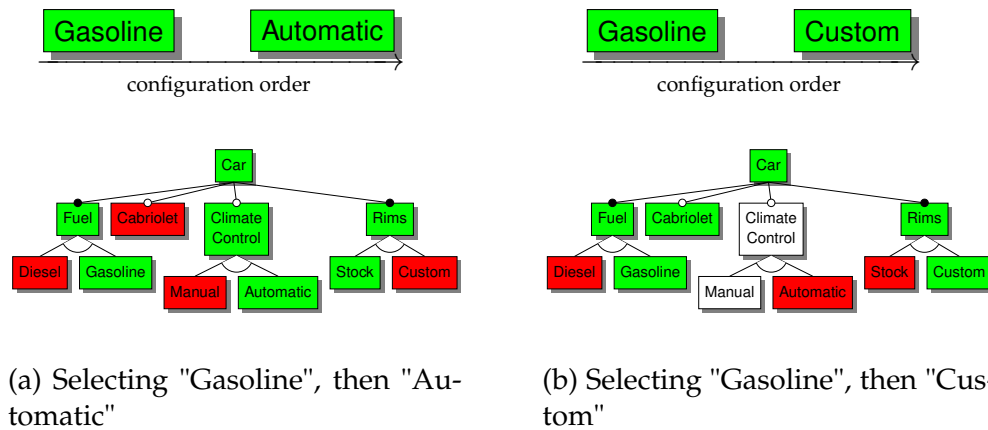


Figure 6.2: Order-dependent outcome in iterative configuration

configuration requires an additional step in which the configuration initially proposed by the stakeholder (i.e., the prototype) is then adjusted as required to ensure validity. Automation must be employed to ensure that potential changes are automatically identified and communicated to the user. In Figure 6.3, we can see an example of prototype configuration for the same configuration and feature model. In this case, the user has decided to specify the desired configuration status of three features simultaneously (for the sake of the example, we chose three features, but any number could have been chosen, from two up to the complete configuration if so desired). These features ("Automatic", "Cabriolet", and "Custom") are all selected in the idealized configuration, but cannot be simultaneously selected in a viable configuration. Therefore, a conflict exists and a conflict resolution step must be taken, where the user will resolve whatever problems may exist in his specified prototype configuration, with the automated help of the system. Support is provided by considering the prototype configuration to be an invalid configuration that must be repaired to conform to the feature. The system will analyze the configuration, identify possible repairs, and presents them to the user so that he may chose his preferred way of resolving the inconsistency.

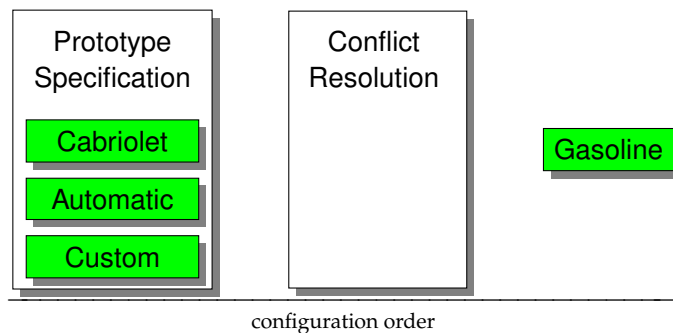


Figure 6.3: Prototype configuration

In the case of Figure 6.3, the system could identify and provide to the user the following options:

- Deselect "Automatic"
- Deselect "Cabriolet" and "Custom"

Which would give the user a clear perspective of the possible trade-offs involved with those features. The problem of identifying the changes required to validate an invalid configuration is not new, and is known in the literature as configuration repair or diagnosis [WBS<sup>+</sup>10, XHSC12, WPX<sup>+</sup>13]. For such an approach to be usable in the context of configuration, the following properties must be met:

1. **High Performance.** The configuration repair approach must be efficient so that the user's experience and workflow is not disrupted by excessive runtime requirements.
2. **Multiple Repairs Identification.** The configuration repair approach must be able to correctly identify variant alternative repairs, rather than a single possibility, so that the user may determine what the best compromise is.
3. **Conciseness.** Although the user must be made aware of repair alternatives, he cannot be overwhelmed by choices.
4. **Applicability to Boolean feature models.** To meet the feature modeling approach used in our work.

We found that existing approaches did not meet one or more of these criteria, so we have proceeded to develop our own. The configuration diagnosis technique proposed in [WBS<sup>+</sup>10] is based on the transformation of the feature model and configuration to be repaired into an equivalent CSP problem. Runtime performance is improved by transforming the optimization problem into a satisfaction problem, by considering a parameter that is a bound on the acceptable quality of the solution. If this bound proves too tight, the process must be repeated with a relaxed bound, until a satisfactory (but not necessarily optimal) solution is found. The algorithm was experimentally tested with synthetic feature models of up to 5000 features. While the approach seems to scale well, it does not discuss the analysis or presentation of multiple alternative optimal repairs. Trade-offs between optimality and performance exist that can be hard to parameterize.

Recent approaches address alternative variability models with increased expressiveness such as Kconfig or CDL [BSL<sup>+</sup>10]. In [XHSC12], the range fix approach is described, where fixes are computed for each unsatisfied constraint including admissible ranges of variation for numeric configuration parameters. The HS-DAG algorithm [GSW89], based on MAX-SAT techniques, is used to compute consistent variable assignment sets. These are then transformed into fix units by repeated application of a set of transformation rules until a fixed-point is reached. This work is extended in [WPX<sup>+</sup>13] to include an adaptive strategy for iteratively identifying preferential fix options based on previous user feedback. The advantage is that the repair list to be considered at each moment by the user is further reduced. A downside is that the process becomes iterative. Because of this, the user does not have a global perspective of the impact of his choices, and conciseness of the repair list is achieved by reintroducing the iterative element that prototype-based configuration intends to avoid. Also, the range fix approach is relevant for parameterized feature models, Kconfig or CDL, but is not for Boolean feature models.

Other repair approaches have also been described in the literature. However, they tend to have domain specific aspects that are not easily transferable to feature model configuration repair. In [NEF03], the xlinkit framework for consistency management is introduced. It includes a first-order logic language for specifying domain constraints over XML documents. Inconsistencies are detected and an n-ary link connecting the inconsistent elements is produced. A repair manager identifies and makes available to the user all the potential repairs. Since not all repairs are ensured to be consistent with the documents' grammar, certain types of repairs can be administratively prohibited so that the documents remain semantically consistent after repair. Interaction between multiple constraints is also not considered, and may require backtracking over attempted repair actions. No analysis of runtime performance or scalability is provided. A similar approach is applied to the repair of UML diagrams in [ELF08], where a set of domain specific fix procedures is created to ensure overall consistency is preserved. [JM11] describes the use of MAX-SAT techniques to identify potential locations of bugs in source code and identify repairs for certain specific categories of errors. None of these approaches is directly applicable to feature configuration repair without major effort or adaptation.

## 6.2 Configuration Repair Overview

Configuration repair entails identifying the necessary changes that transform an invalid original configuration into a valid target repaired configuration. These changes are indications to toggle the selection status of a feature from selected to deselected (or viceversa). A single repair may include changes to the selection state of multiple features. Since any valid configuration can be designated as the target of the repair, there are as many repairs of a given configuration as the number of valid configurations of the corresponding feature model. Consequently, any non-trivial repair mechanism should not only identify just any one of the potential repairs, but also conduct the optimization of some quality criterion. Typical criteria include the minimization of the Hamming distance between the original and target configurations, effectively selecting repairs that minimize the overall number of changes to feature (de)selection. Other criteria may favor preservation of selected (but not deselected) features, or the minimization of a weighted sum of selected features (e.g., a total cost of product). As an example, consider the following configuration of the feature model in Figure 6.4, adapted from [WBS<sup>+</sup>10], with selected features "Automobile", "Brake Control Software", "Brake Control ECU", "Non-ABS Controller" and "1 Mbits/s CAN bus", and all other features deselected. This configuration is not valid because one of the hard

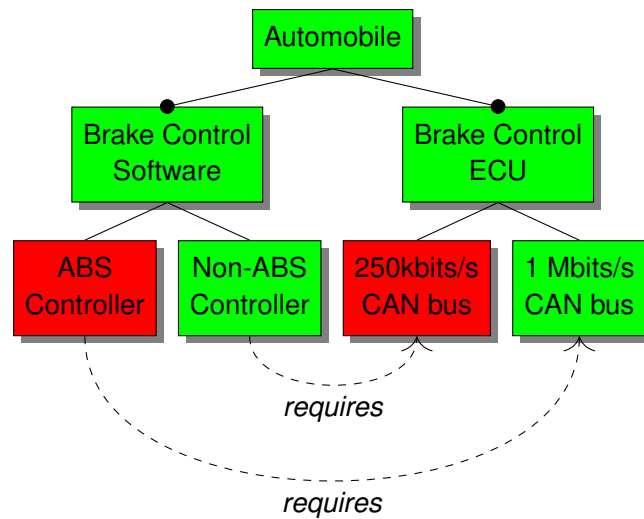


Figure 6.4: Configuration repair example

constraints ("Non-ABS Controller" *requires* "250kbits/s CAN bus") is not satisfied. The two repairs minimizing the Hamming distance [Ham50] between original and repaired configurations are:

- Select "250kbits/s CAN bus"; Deselect "1 Mbits/s CAN bus"

- Select "ABS Controller"; Deselect "Non-ABS Controller"

However, the user may find that one of the possible repairs (such as downgrading to a non-ABS controller) is not really desirable. This only reinforces the necessity of considering all potential repairs. In fact, if the repair tool provides a single repair solution, there is no warranty that the desired solution will be the one outputted. This problem is exacerbated as the complexity of the feature model and number of potential repairs increase. Therefore, generally speaking, all potential repairs should be considered. Efficient ways of identifying, accessing and presenting the alternative repairs have to be considered.

### 6.3 Configuration Repair Based on Cover Information

Our configuration repair algorithm is based on computation of a *cover* of the feature model expression. A cover is a representation of a Boolean function as a disjunction of terms (see Section 6.3.1). The algorithm involves the following steps:

1. **Feature model partitioning and cover extraction.** First, the feature model is partitioned into a set of independent trees for efficiency purposes and also for allowing a concise representation of results. The cover is then computed by analysis of these partitions (Section 6.3.2).
2. **Repair identification (based on cover information).** Possible repairs are identified by comparative analysis of the variable assignments in the invalid configuration with those required for satisfying each cover term. The algorithm is ensured to identify the optimal repair in terms of Hamming distance, but other metrics may also be used, in which case a best-effort attempt will be made (Sections 6.3.3-6.3.6).
3. **Result presentation.** The partitioning approach allows not only efficiency gains, but it also provides a natural way for decomposing the possible repairs into independent problems that can be individually resolved. This allows a much more concise representation of the repairs, with corresponding benefits for the user. The system can also iterate efficiently through all possible repairs, if required (Section 6.4).

The first step must be performed only once per feature model, while the latter two must be performed once per configuration to repair (or when a new metric

is defined). This benefits the performance of our algorithm, since cover computation is the most computationally expensive task.

The prototype tool for configuration repair is presented in Section 6.5.

### 6.3.1 Cover and Literal Minimization

Our configuration repair algorithm is based on extraction of the *cover* of the feature model expression. While an expression in conjunctive normal form (CNF) represents a Boolean function as a conjunction of disjunctions, a cover is a representation of a Boolean function in the format of disjunction of terms, with a term being a conjunction of literals (variables or their complement). Accordingly, a cover of function  $f$  of Boolean variables  $x_i$  can be represented as:

$$f(x_1, x_2, \dots) = \bigvee_i T_i(x_1, x_2, \dots) \quad (6.1)$$

Where  $T_i(x_1, x_2, \dots)$  is a product of some or all of  $x_i$  or their complement  $\neg x_i$ . All Boolean functions can be expressed as a cover (or CNF form). For convenience, we refer to any cover equivalent to a function  $f$  as an *ON-cover* of  $f$ . In opposition, the *OFF-cover* of function  $f$  is equivalent to the complement of  $f$ . A minimal cover has a minimal number of terms. Literal minimization ensures that the terms do not include redundant unnecessary variables. Cover and literal minimization is a hard problem, but of critical importance in the areas such as logic circuit design. As such, many heuristic algorithms have been proposed throughout the years such as Karnaugh Maps [Kar53], the Quine-MCcluskey algorithm [Qui52, Qui55, McC56, NNCI95], and the Espresso heuristic logic minimizer [Rud86]. While the former two have scalability issues and can only tackle problems of modest dimension, the latter is the state-of-the-art in logic minimization, and includes numerous hybrid heuristic techniques able to successfully handle complex logical function minimization with virtually unlimited number of variables. It is also capable of performing multiple-valued logic optimization. The reader is referred to [Rud86] for additional details.

### 6.3.2 Feature Model Partitioning for Efficient Cover Computation

A feature model may be converted to an equivalent Boolean expression according to the transformations described in [Bat05]. Although the Espresso heuristic logic minimizer is very efficient performance-wise, cover minimization is a very

hard problem and a strategy based on direct processing of the feature model expression is destined to failure. Therefore, we have developed an approach based on a partitioning strategy that decomposes the feature tree into multiple independent partitions. These partitions can be independently analyzed and the results combined, achieving very efficient processing of large feature models. Our partitioning approach includes three phases. These will be described using the feature model in Figure 6.5 as an example.

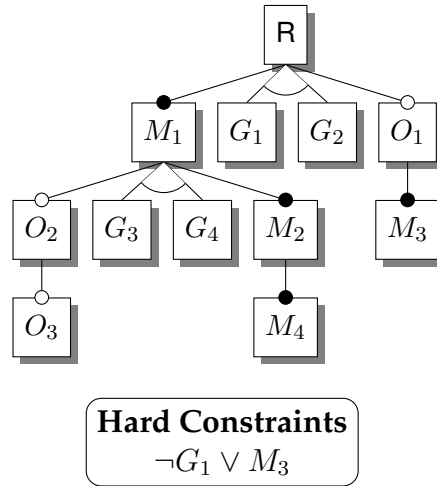


Figure 6.5: Feature model decomposition example

In the first phase, illustrated in Figure 6.6, the feature tree (not model) is decomposed into a set of multiple independent trees, using Algorithm 4.

Decomposition is recursive, but only on mandatory features. This is because the selection state of features that descend from root along a mandatory path is well-known (they must be selected in valid configurations). Therefore, these features do not constrain in any way the selection of their children. In other words, the selection state of each child of a feature with mandatory path to root can be considered independently of the selection state of their parent (which is always selected). In contrast, the selection state of any child of an optional feature is conditioned by the selection state of the parent: if the parent is not selected, then

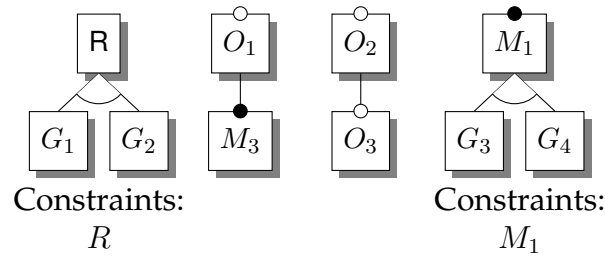


Figure 6.6: Feature model decomposition example - Step 1

**Algorithm 4:** Partition feature tree

---

```

1 Function partitionFeatureTree
  input : Feature tree with root  $f$ 
  output: Partitioned feature tree

2  treeSet  $\leftarrow []$ 
3  foreach mandatory children  $m$  of  $f$  do
4    Add partitionFeatureTree ( $m$ ) to treeSet
5  foreach optional children  $o$  of  $f$  do
6    Add subtree rooted at  $o$  to treeSet
7  foreach children group  $g$  of  $f$  do
8    Create feature tree  $z$  with root  $f$  and single child  $g$ 
9    Add to  $z$  constraint requiring selection of  $f$ 
10   Add  $z$  to treeSet
11 return treeSet

```

---

the children may not be selected. Therefore, it is possible to proceed recursively and consequently ignore parent features only while traversing a mandatory path from the root. Groups are extracted from the original tree along with the parent feature.

After this step, the feature tree in Figure 6.5 would be decomposed into the four sub-trees as seen in Figure 6.6. Features  $M_2$  and  $M_4$  have disappeared due to the recursive descent of the partitioning algorithm.

In the second step, variables that were removed (disappeared) after the first step are reintroduced into the decomposed feature tree set as a new feature tree including only a single root node corresponding to the removed variable and a constraint requiring its selection (see Figure 6.7). This is necessary because some mandatory features of the original model will not be found in any one of the decomposed trees. Although, in fact, mandatory features are sometimes preprocessed out of feature models before processing to improve performance [Men09], since they do not include any meaningful variability information, we believe this

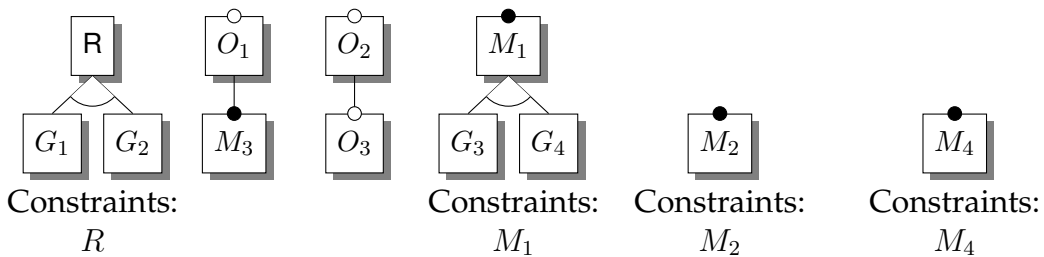


Figure 6.7: Feature model decomposition example - Step 2

approach is not well suited for our problem, since a configuration may be rendered invalid precisely because of failure to select a mandatory feature. If the corresponding variables were removed, no appropriate advice could be offered in those cases. Additionally, since selection of the root of the decomposed trees is not implicit (unlike in the original feature model), a constraint is also added that ensures the mandatory selection of the root  $R$ .

In the third step (illustrated in Figure 6.8), the domain of the constraints of the original feature model is analyzed to determine if some of the decomposed trees should be merged back into a single tree. Constraints are then added to the corresponding trees (that is, the tree including its domain variables). Feature trees are merged by creating a common anonymous parent whose purpose is simply conjoining the child feature trees. The feature tree expression of two or more merged feature trees will be the conjunction of their individual feature tree expressions. The outcome of this process will be a set of constrained feature trees<sup>1</sup>. These constrained feature trees are to be independently analyzed by the heuristic logic minimizer in ensuing steps of the repair algorithm.

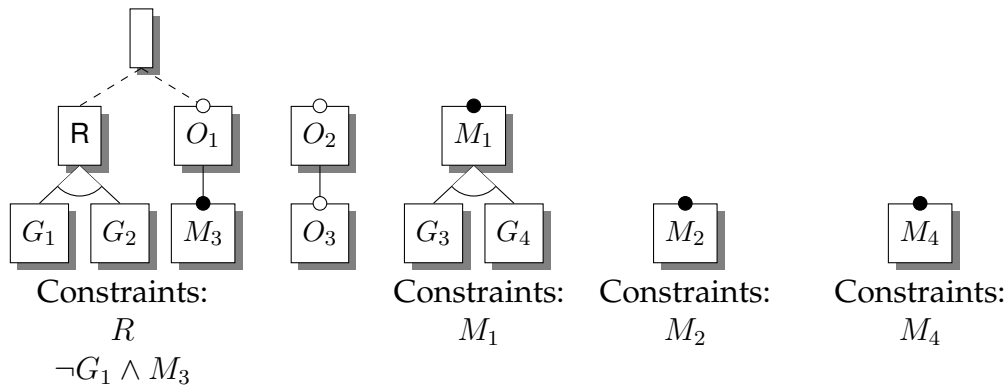


Figure 6.8: Feature model decomposition example - Step 3

### 6.3.3 Configuration Repair Using Cover Information

A repair represents the difference between an invalid original configuration and a valid target configuration. On the other hand, a term in a cover describes one or more valid configurations. Therefore, given a cover and a configuration to be repaired, it is possible to compute the repairs by calculating the difference between each term in the cover and the original invalid configuration. As an

<sup>1</sup>Although these are remarkably similar to a feature model, there are some differences so we use a different terminology to avoid confusion. Differences include the possibility of having optional nodes at root, whose selection is not mandatory unlike in standard feature models. Also absent from standard feature models are the anonymous nodes at the root of the merged trees.

example, consider a feature model in Figure 6.9, with features  $\{A, B, C, D, E, F\}$  and the cover given by Equation 6.1 with

$$T_0(A, \dots, F) = A \wedge \neg C \wedge \neg D \wedge \neg E \wedge F \quad (6.2)$$

$$T_1(A, \dots, F) = A \wedge C \wedge E \wedge F \quad (6.3)$$

$$T_2(A, \dots, F) = A \wedge C \wedge D \wedge F \quad (6.4)$$

The first term of the cover will be satisfied by any configuration including fea-

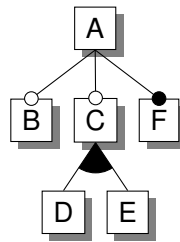


Figure 6.9: Feature model for example of repair extraction from cover terms

tures  $A$  and  $F$ , that not include features  $C$ ,  $D$ , nor  $E$ . The second term is satisfied by any configuration including  $A, F, C$  and  $D$ , while the third term is satisfied by inclusion of  $A, F, C$  and  $E$ . If a configuration is valid, then it must by definition satisfy at least one of the terms, since the cover is logically equivalent to the feature model expression. Therefore, any configuration satisfying any one (or more) of these three terms will satisfy the cover expression and will be a valid configuration. This means that it is possible to compute all possible repairs by finding the changes to the current configuration that are required to satisfy each and every one of the terms in the expression (if a single repair is sought, then it is sufficient to find the changes that validate any single term). If open (undetermined) variables still exist in the configuration, they can be safely disregarded: we can assume these will be configured appropriately in the future to obtain a valid configuration. This can either be enforced by the configurator (see Chapter 5), or, alternatively, the repair algorithm can be run again if subsequent configuration errors arise. For example, consider the invalid configuration represented in Figure 6.10.

- Selected  $\{A,B,C\}$
- Deselected  $\{D,E\}$
- open  $\{F\}$

The following repairs can be identified after analysis of each term:

- **Repair from first term:** Deselect  $\{C\}$
- **Repair from second term:** Select  $\{E\}$
- **Repair from third term:** Select  $\{D\}$

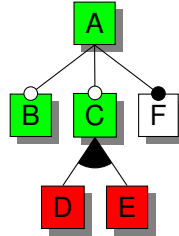


Figure 6.10: Invalid configuration

The first repair is obtained by taking the first term  $T_0(A, \dots, F)$  and computing the variation of non-open configured variables required to satisfy it. The term is currently unsatisfied because of the selected status of  $C$ . Features  $A$ ,  $D$ , and  $E$  are properly configured, while  $F$  is open and can be assumed to be eventually configured selected as required. Therefore, the configuration can be fixed by changing variable  $C$  to 'deselected'. The second and third repairs are obtained by conducting a similar analysis based on the remaining terms  $T_1$  and  $T_2$ . Algorithm 5 describes the process by which a term is analyzed to compute the repair for a given configuration.

---

**Algorithm 5:** Computing a candidate repair from a specific term
 

---

```

1 Function generateTermRepair
  input : Configuration config, term term
  output: A repair of configuration config, allowing satisfaction of term

2  repairOutput  $\leftarrow$  []
3  foreach conjoined element  $e \in (x)$  of term evaluating to false in config do
4    /* By definition of term,  $e(x) = x$  or  $e(x) = \neg x$  */
    Add  $x$  to repairOutput
    /* repairOutput contains all variables that must be changed in
       config to satisfy term */
5  return repairOutput
  
```

---

Considering a term is a conjunction, all conjoined elements that integrate it must be satisfied to ensure satisfaction of the term. Thus, the solution identified by the Algorithm 5 is unique and therefore optimal with respect to the minimum number of changes required to satisfy the input term. Considering a cover is a disjunction of terms, each repair satisfying one such term also satisfies the cover

itself. Therefore, we identify repairs by iterating over all terms of a cover, producing a set of candidate repairs for the cover (Algorithm 6). Each such candidate repair is associated with the satisfaction of a specific cover term.

---

**Algorithm 6:** Computing all candidate repairs

---

```

1 Function getCandidateRepairs
  input : Configuration config, cover c
  output: A set of candidate repairs
2   candidateRepairs  $\leftarrow$  []
3   foreach term t in cover c do
4     Add generateTermRepair (config,t) to candidateRepairs
5   return candidateRepairs

```

---

### 6.3.4 Performance and Optimality

Computation of the candidate repairs, described in Section 6.3.3, requires iteration over all the terms in a cover. Therefore, runtime performance of this step of the algorithm is dependent on the total number of terms. The worst-case scenario corresponds to a cover where all terms include all variables or their complement. In this case, each term fully specifies the selection state of all features corresponding to a single valid configuration, so there will be as many terms as possible configurations. Computing the repairs using the approach in Section 6.3.3 would be prohibitively expensive. However, in all likelihood that cover is not minimal. Conversely, the best case scenario is obtained when a minimal cover is used, since the least number of terms have to be analyzed. Therefore, using the minimal cover for repair identification using the above process will achieve optimal efficiency of this step.

Since candidate repairs are computed considering each term of the cover individually, not all the repairs identified using the approach outlined above are necessarily unique, orthogonal or minimal with respect to the full cover expression. These are simply candidate repairs and must be subjected to further selection as described later. However, it is ensured that at least one repair with minimal number of changes is in fact included among the candidate repairs. To prove this by *reductio ad absurdum*, let  $T$  be the set of all terms in the cover,  $R(t) : t \in T$  the repair obtained from term  $t$  with algorithm 5 and  $C(r)$  the number of configuration variable changes described in repair  $r$ . Consider as an hypothesis that an optimal repair  $H \notin T$  exists for which  $C(H) < C(R(t))$  for all  $t \in T$ . We know

from the discussion in Section 6.3.3 that  $R(t)$  is the unique solution that specifies the absolute minimum number of changes required to allow satisfaction of  $t$ . Then, consider that all valid configurations must, by definition, satisfy the feature model expression (and any equivalent cover). Also, a cover is a disjunction of all  $t \in T$  so it is satisfied iff at least one term is also satisfied. Therefore, if  $H$  is a valid repair it must transform the defective configuration into one that satisfies at least one term  $t_s \in T$ . By hypothesis,  $H$  specifies less variable changes than those required by  $R(t_s)$ , that is  $C(H) < C(R(t_s))$ . However, this is not possible, because  $R(t_s)$  specifies the minimum number of changes required for satisfying  $t_s$ . Therefore, the hypothesis is demonstrated to be false: there is no repair  $H \notin T$  for which  $C(H) < C(R(t))$ . It is interesting to observe that, to obtain this result, optimality of the cover is not required. Therefore, cover minimization is not required for generating minimal repairs, although it can be advantageous as it reduces the total number of candidate repairs that are generated. Correspondingly, all experiments described in this work are based on minimal covers.

### 6.3.5 Selection Criteria

As mentioned earlier, further selection is required for identifying the most interesting options among all the candidate repairs. This is achieved by specifying some metric that identifies preferable repairs. The most common and natural choice is perhaps the Hamming distance [Ham50] between the defective and repaired configurations (i.e., the number of changed variables). In this case, as demonstrated in Section 6.3.4, optimal results are ensured. However, other metrics can be considered to allow for alternative repair selection strategies. In this case, optimization is not ensured over all the configuration space, but only over the space of the repaired configurations attainable from application of all repairs  $R(t) : t \in T$ . Although global optimization is not achieved in this case, selected repairs will be biased accordingly, allowing best effort selection of repairs with desired properties.

### 6.3.6 Repair of Partitioned Feature Models

Section 6.3.3 discusses the approach for repairing using a single cover expression. In this section, we discuss how this basic process can be used to obtain repairs for a feature model partitioned according to the algorithm described in Section 6.3.2.

The ON-covers of the feature model partitions are first obtained using the

Espresso logic minimizer. This requires the initial specification of an ON- or OFF-cover. The feature model expression obtained by the transformations described in [Bat05] tends to be closer to clause normal-form (CNF), rather than sum-of-products (SOP). Since obtaining the ON-cover from straightforward conversion from CNF to SOP may result in exponential explosion of the number of terms, we instead generate the OFF-cover instead, which can be efficiently computed by De Morgan's law [De 58], complementing each clause and disjoining the results.

The ON-covers of the partitions can be combined to obtain the ON-cover of the original, non-partitioned, feature model. Terms of different partitioned trees cannot be simplified when conjoined, due to variable independence, so the direct conjunction will yield the minimal cover of the original feature model. This means that it is sufficient for our purposes to compute the covers for each individual partition. Rather than explicitly compute the conjunction, however, we use an index-based system, using *repair lists*, to account for all possible term permutations of interest that might be generated.

Considering that terms of the ON-cover of the original non-partitioned feature model are permutations of the terms of the ON-cover of each individual partition (these will be designated as partitioned terms in the sequel), the overall number of terms in the minimal ON-cover of the non-partitioned feature model  $\#T_F$  can then be computed as:

$$\#T_F = \prod \#T_p \quad (6.5)$$

Where  $\#T_p$  is the number of terms of the minimal ON-cover of partition  $p$ . Rather than explicitly generating all these permutations, however, the process of repair identification described in Section 6.3.3 is applied individually to each partition, and the repair for the complete feature model is then derived from those results. Therefore, we generate the candidate repairs for each partitioned tree by traversing all partitioned terms, but not their permutations. Since the overall number of partitioned terms  $\#P$  is given by:

$$\#P = \sum \#T_p \quad (6.6)$$

Performance improvement over analysis of all terms of the original feature model is given by:

$$\frac{\#T_F}{\#P} \quad (6.7)$$

This is a significant factor in practice as experimental results will demonstrate. The runtime and space performance of the algorithm is generally proportional to

$\#P$  rather than  $\#Tp$ .

Repairs for the non-partitioned feature model can be obtained by considering permutations of repairs of the individual partitions. However, the process of repair generation described above may generate duplicate repairs. For example, the partial configuration

- Deselected {A}
- Open {B,C}

with cover  $A \wedge B \vee A \wedge C$  will generate two identical repairs:

- **Repair from first term:** Select {A}
- **Repair from second term:** Select {A}

Duplicate solutions are disregarded in each partition before permutations are considered, eliminating duplicated repairs and further improving algorithm performance when iterating over all possible repairs. All the possible repairs for the non-partitioned feature model may be efficiently represented as a repair list containing, for each partition, the indexes of one of each term that generates a unique repair. For example, if a model is split into two partitions, the first with a 6-term cover and the second with a 10-term cover, then the repair list  $\{\{1,3\},\{1,4,5\}\}$  indicates that only the terms 1 and 3 of the first partition generate unique repairs, and similarly for terms 1, 4 and 5 of the second partition. Other terms in the same cover would produce similar repairs and can be disregarded. The six possible repairs of the non-partitioned model would correspond to all the permutations of the repairs corresponding to terms  $\{1,3\}$  of the first partition and  $\{1,4,5\}$  of the second partition. A specific repair can then be represented as an iteration list, containing indexes of the repair list. For example, assuming 0-based indexes, repair  $\{1,1\}$  corresponds to the repair obtained by combining the repair generated from term 3 of the first partition with the repair obtained from term 4 of the second partition. This allows efficient iteration and random access over all possible repairs. While all repairs can be generated and iterated over with this method, usually repairs are obtained with a specific optimization criterion in mind. A quality criterion depending on the original term and repair (or equivalently, depending on the repair and targeted configuration) can be computed in tandem with repair generation, so minimization can be achieved by exhaustive search over all generated repairs. If the criterion is composable according to the partitions found, then this also allows optimization for the non-partitioned feature

model. For example, minimization of the Hamming distance between the original and repaired solution can first be performed individually for each partition (and results then combined to obtain the corresponding value for the original non-partitioned feature model). While this is only possible if the quality assessment metric is composable across partitions, we find that this is the case for common and plausible quality metrics, such as the Hamming distance or any other linear combination of weighted feature selection variables or toggle status (which allows, for example, creating criteria that favor repairs preserving selected features or similar metrics). Partitions can also be merged to allow for non-composable metrics to be used. Nevertheless, it should be pointed out that, as discussed in Section 6.3.3, the search space includes only each term repair generated from the cover analysis. Therefore, minimization of the evaluation criterion is conducted only over this space. Evaluation of the quality criterion allows further refinement of the repair list described above, as only repairs with optimal evaluation need to be represented. Continuing the example, if only term 1 generates an optimal repair in partition 1, while partition 2 is optimized by the repairs of terms 1 and 5, then the repair list can be further reduced to  $\{\{1\},\{1,5\}\}$ , representing the two optimal repairs of the non-partitioned feature model.

The minimal cover must be computed only once for each feature model. However, alternative criteria do not require cover recomputation. Construction of the repair list must be conducted only once for each repair of a specific configuration with some optimization criteria of that model. Subsequent iteration over all alternative repairs, or random access to specific optimal repairs, requires only proper update of the iteration list (and derivation of the repairs associated with the respective terms), making iteration over all alternative repairs extremely efficient.

## 6.4 Presentation of Potential Repairs

As demonstrated by the experimental results of Chapter 7, large numbers of alternative optimal solutions may be found by the previously described techniques. Settling for just any single one of such solutions is sub-optimal in the sense that the user might prefer one of the undisclosed solutions. Although an exhaustive listing of all possibilities may be suitable for very small feature models, that is not true in all cases, as the number of potential repairs may be in the order of thousands, millions or even higher, for feature models of practical dimension.

We take advantage of the partitions identified by the process described in Section 6.3.2 for decomposing the required repairs into independent *problems* that

should be resolved to repair the configuration. In this context, a problem corresponds to a situation where a repair is required for one (or more) of the independent partitions. Resolving one of these problems entails finding the appropriate repairs for the corresponding partitions. Generically speaking, the repair of a damaged configuration can be decomposed as the resolution of several problems. These problems are initially created with the following process:

- One problem is created for each feature model partition corresponding to an element of the repair list with cardinality 2 or higher.
- One additional problem is created including all the repairs associated to elements of the repair list with cardinality 1.

The former problems correspond to repair decisions of feature model partitions that involve the selection of one or more alternative repairs. An example would be the mandatory selection of one of multiple features in a 1..1 cardinality group. The latter problem is an aggregation of all "mandatory" repairs that do not have valid optimal alternatives. An example would be the obligatory selection of mandatory children of the root of the feature model. It is mostly of informational nature, as the user is provided with no option other than accepting the proposed changes, unless validity or optimality is to be sacrificed. The repairs associated to each one of these problems can be easily obtained and iterated over by considering a repair list including only the relevant partitions. By decomposing a repair into the resolution of multiple independent problems, the combinatory explosion resulting from the combination of the multiple alternatives is avoided. This scales better and is more manageable than the alternative selection from an exhaustive list including all possible permutations.

## 6.5 Tool Description

We have implemented our algorithm in the Java programming language and created a supporting tool. The API allows the user to specify a functor that will be used to evaluate repairs. New evaluators implementing the required interface may be easily created by the developer, allowing flexibility in defining metrics alternative to default Hamming distance. Access to repairs is offered by returning an iterator that traverses all optimal repairs. This iterator can be configured to consider only repairs for a specified selection of partitions, enabling the problem-based presentation outlined above. We implemented a tool based on this API to

demonstrate the algorithm and conduct experiments. The user is able to load feature models in the format of the SPLOT feature model online repository, specify a candidate configuration, partial or complete, and see the required repairs. Figure 6.11 shows our tool being applied to the home integration system (his) feature model [KLD02].

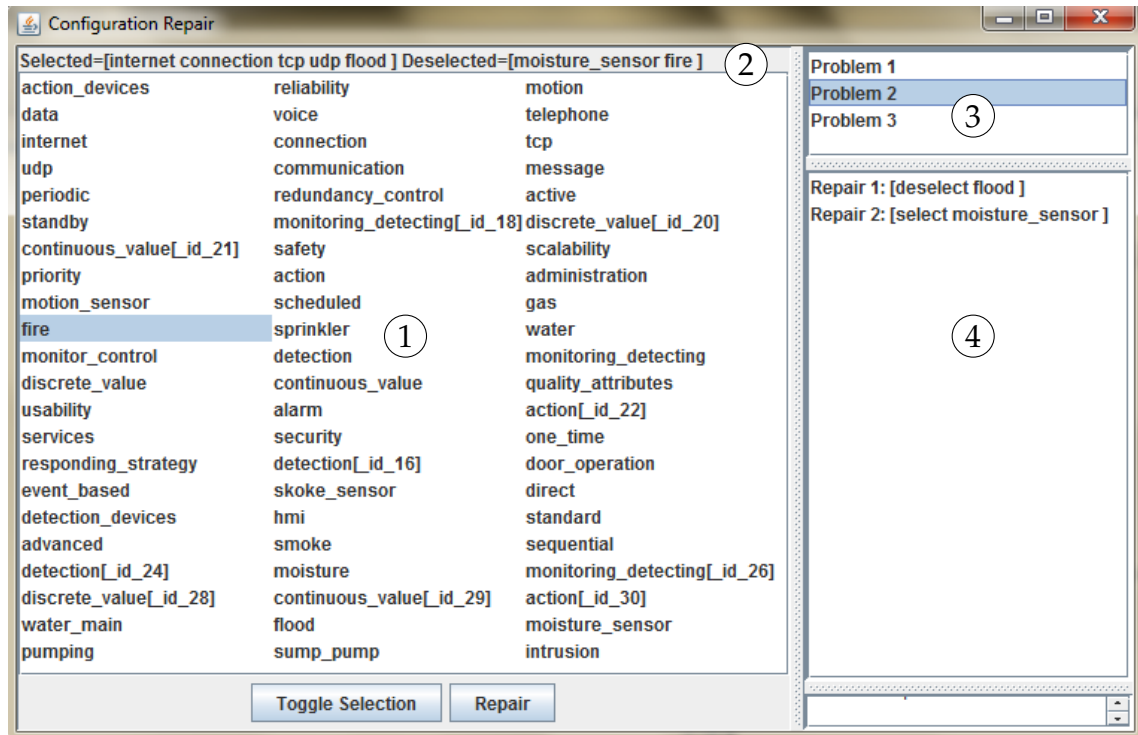


Figure 6.11: Configuration repair tool in action

The center left box ① displays the features of the Home Integration System (his) feature model. The user is able to select/deselect these features at will, and has currently selected the internet, connection, tcp, udp and flood features and deselected the moisture sensor and fire features, as seen in the label displayed above the feature list ②. On the top right, the user can select one of the three problems found ③. Available repairs, minimizing the Hamming distance, are presented in the bottom right box ④. The partial configuration specified by the user and seen on the label at the top ② has three defects:

1. The connection feature requires one and only one of http and udp.
2. The flood feature requires the moisture sensor feature.
3. The fire feature is a mandatory child of the root.

These problems are independent and can be separately presented and resolved by the user. It can be seen, on the top right box ③, that the application successfully achieves this. On the box bellow (right hand side) ④, the application shows the two alternative repairs available for the problem two, which is currently selected:

1. Deselect flood
2. Select moisture sensor

Repairs for each other problem, not represented in the figure, are "Deselect http" or "Deselect udp" for the first problem and "Select fire" for the third.

## 6.6 Conclusions

Once the stakeholder decides upon an idealized configuration (whether it is fully or just partially complete), it must then be realized into an actual, valid implementation. If the idealized configuration is also valid, then no compromise is necessary and the idealized and realized configurations are identical. However, that is not necessarily the case: the idealized configuration may be inconsistent and some differences with respect to the realized configuration must exist. These differences correspond to trade-offs that were necessary to achieve feasibility. Nevertheless, alternative choices and different resolutions may be possible: the realized configuration may be obtained by deviating from the idealized in many different, alternative ways. Each such way represents a different form of compromise, or trade-off, which may have different levels of usefulness or desirability for the stakeholder.

Standard configuration techniques don't address the problem of helping the user to decide or find the best compromise. Iterative configuration approaches require that the configuration is specified on a step-by-step basis, one feature at a time. This allows the configurator to propagate each choice as required to ensure validity is preserved. However, the result becomes order dependent. The stakeholder has no explicit way of directing the solution towards a desired outcome, or deciding upon the preferable trade-offs.

Prototype-based configuration generalizes iterative configuration to allow more than one feature to be specified in each iteration step (it is possible to consider even the scenario where the entire configuration is specified in a single pass). The major advantage is that an invalid set of choices may be chosen in this way, whereas in standard iterative approaches, invalid choices cannot be represented at all, precluding reasoning and improved support. The point of prototype-based

configuration is that, by supplying the system with an explicit representation of the stakeholder's desires, it becomes possible to analyse them and provide valuable feedback. This allows the stakeholder to make a better and more informed choice about the best way to proceed with the configuration (that is, what trade-off to make).

Each iteration of prototype-configuration includes two steps. First, the intended configuration is provided to the system (i.e., the prototype is specified). Then, if the prototype is valid, no further action is required and the next configuration step ensues. However, if the prototype is not valid, then the system automatically identifies what are the possible changes (or repairs) that can be made to it to ensure validity is attained. The system will actually identify the trade-offs and the stakeholder will explicitly select the option he deems more interesting.

Identifying configuration repairs is a well-know problem with an existing body of work. However, to be suitable to our approach, it must meet some strict criteria that are not all met by current proposals, in terms of performance, suitability for our modeling approach, capability of identifying multiple alternative repairs and also offering a concise view of repair possibilities. We have therefore developed a cover-based configuration repair algorithm that is well suited for our own purposes.

Our configuration repair approach is based on the computation of the cover of the feature model. We partition the feature model for efficiency purposes and use state-of-the-art optimization algorithms to extract the cover. This is the most computationally expensive operation, but is required only once per feature model, rather than once per repair. After that, a set including all optimal repairs, according to the Hamming distance, can be computed efficiently. A repair index list approach allows the repairs to be computed without recombining all the individual partitioned models, with high efficiency gains. Other optimization metrics are also possible, on a best-effort basis. A prototype tool was implemented, for demonstration and validation purposes.





# Validation

We resorted to several different approaches for validation of our work, from automated test case generation up to empirical testing with human subjects. Testing is primarily based on feature models available from the public repository (S.P.L.O.T.)<sup>1</sup>, although to serve the purposes of our tests these were manually or automatically augmented with soft constraints, as described in the corresponding sections. Unless stated otherwise, all tests were conducted equipped with a laptop with a 2.0GHz processor, 8 Gb RAM and the Windows 7 operating system.

Box and whisker plots are often used throughout this chapter to represent the results<sup>2</sup>. While this type of plots is often used to compare alternative series of data, our main purpose here is to represent the distribution of results and identify outlier results.

We conducted tests for assessing the performance of our algorithms for detecting suspicious soft constraints interactions, as well as assessing the prevalence of these interactions (Section 7.1). The configuration repair algorithm was also similarly tested (Section 7.2). We also conducted an empirical assessment of enhanced configuration (Section 7.3).

---

<sup>1</sup><http://www.splot-research.org/>

<sup>2</sup>The reader is referred to Appendix B for a brief presentation of the concepts and symbology

## 7.1 Identification of Suspicious Interactions

In Section 4.3 we defined and described the procedure for identifying potentially anomalous interactions, specifically unsatisfiable, untriggerable and contradictory soft constraints. To recapitulate;

- **Unsatisfiable soft constraint.** This is a soft constraint that is always unsatisfied, in any valid configuration.
- **Untriggerable soft constraint.** This is a soft constraint that may be satisfied only if untriggered.
- **Contradictory constraints.** This is a set of soft constraints that cannot be simultaneously satisfied while simultaneously triggered.

We conducted a series of tests destined to assess both the performance and prevalence of these interactions. This section discusses the goals of these tests, the construction of the data sets used in these experiments, as well as the soft constraint injection algorithm. Section 7.1.3 describes the experiment conducted to assess unsatisfiable and untriggerable soft constraints, while contradictory constraints are addressed in Section 7.1.4.

### 7.1.1 Experiment Objectives and Goals

In this experiment, we seek to investigate the following issues, to address research questions 1(a)i and 1(a)ii:

- **What is the performance of the identification techniques?** Having a clear notion of the performance of the identification techniques is relevant when considering their intended application. They are to be integrated into feature modeling edition tools that support annotation with soft constraints, providing real time feedback as new soft constraints are added by the domain engineer to the model. Therefore, it is important that the performance profile is consistent with this type of application. As reference values, we resort to research in human-machine interface that identified some important thresholds for response delay [Mil68, CNM83, New90, CRM91]. In particular, 0.1 [s] is the threshold for maintaining the illusion of continuous unimpeded operation, making the user essentially oblivious to machine operation. Longer delays up to 1[s] become noticeable but are well tolerated: the user notices that the machine is working, but finds the delay acceptable and

the sense of flow is not broken. 10 [s] is the maximum time limit that still allows the user to focus his attention on the task, however, progress indicators and some means for interrupting the task should be provided. Delays over 10 [s] are only acceptable in natural breaks in the flow of work, such as switching to a new task.

- **What is the prevalence of these suspicious interactions?** Since soft constraints are currently not in widespread use, it can be difficult to estimate just how often these interactions come into play without concrete tests. This information can be combined with the performance assessment to make a better judgement on whether or not it is advisable to include tests for specific types of interactions. For example, it might not be reasonable to include an expensive test to detect a very rare interaction. On the other hand, it might be justifiable to test for the same interaction, in spite of its rarity, if testing performance is very high.

### 7.1.2 Data Set Construction and Constraint Injection

To obtain the data set necessary for these experiments, we relied on feature models publicly available at the S.P.L.O.T. online repository [MBC09]. These were supplied by the site users and include models from both academic and industrial origin. Tapping into these resources provided us with a large set of non-synthetic non-random feature models with diverse characteristics and relevant dimension. From all models available at time of access (Feb. 2014), we decided to use all those with 40 or more features. This resulted in selection of the models presented in Table 7.1 to be included in our input data set. The CTCR (Cross Tree Constraint Ratio) and clause density parameters found in the table are important measures for assessing the difficulty of solving the satisfiability problem of the feature model expression [MAK09]. CTCR is the ratio between features in constraints and total number of features, while clause density is the ratio between the number of clauses and features.

One problem we had to overcome is that the models in the S.P.L.O.T. repository are not annotated with soft constraints. While we could provide these annotations for some models, the wide range of domains and conflict of interest advise against that option. Therefore, we decided to resort to automatic annotation. Rather than generating completely random annotations, we used the prototypical patterns of application of soft constraints described in Section 4.2 to guide

Table 7.1: Feature models included in input data set.

Model Name	Features	Optional	Groups	Hard Constr.	CTRC	Clause Density
AndroidSPL	45	8	9	5	17%	0.6
Arcade Game PL	61	5	9	34	55%	1
BankingSoftware	176	77	16	4	2%	0.8
BattleofTanks	144	15	10	0	0%	
bCMS system	66	12	8	2	4%	0.7
Billing	88	45	2	59	65%	1
Car	72	10	19	21	31%	0.8
Coche_Ecologico	94	12	16	2	4%	0.5
Consolas de Videojuegos	41	2	11	5	21%	0.6
OS	40	7	7	0	0%	
DATABASE_TOOLS	70	20	7	2	8%	0.3
DELL Computers	46	1	8	110	76%	2,9
Documentation_Generation	44	3	9	8	29%	0.6
DS Sample	41	0	6	0	0%	
Eclipse1 - Reuso	72	40	7	1	2%	0.5
Electronic Drum	52	1	11	0	0%	
Estrutura_Decisiones <sup>a</sup>	366	2	19	192	93%	0.6
E-science application	61	7	16	2	4%	0.7
Face Animator	40	8	11	17	25%	1,7
FraSCAti	63	39	2	46	57%	1,3
HIS	67	10	6	4	11%	0.5
Hotel Product Line	55	32	7	0	0%	
J2EE web architecture	77	27	11	0	0%	
Jogo	59	8	14	0	0%	
Letovanje	43	3	13	2	6%	0.7
Linea de Experimentos	52	20	4	4	0%	
Model_Transformation	88	12	25	0	0%	
MFP	56	5	8	90	66%	2,4
Meeting Config	57	13	9	0	0%	
Smart Home (a)	56	36	4	0	0%	
Smart Home (b)	60	30	6	2	6%	0.5
SmartHome_vConejero	59	33	0	3	6%	0.8
Thread	44	15	7	0	0%	
Video Player (a)	53	17	9	2	7%	0.5
Video Player (b)	71	12	5	0	0%	
Webmail	81	29	7	0	0%	
Web_Portal	43	17	6	6	25%	0.5
Xerox	172	1	28	0	0%	
xtext	137	95	0	1	1%	0.05

<sup>a</sup> A Model for Decision-Making for Investments on Enterprise Information Systems

the automatic annotation process. These patterns are the OSS (Optional Selection Sugestion), RCS (Reversed Constraint Suggestion) and GSS (Group Selection Suggestion). We believe that by generating annotations using these patterns as guidelines, we may obtain data that is more meaningful than that obtained from a completely random approach. Nevertheless, since soft constraints are not in widespread use in feature modeling, it can be hard to ascertain just how representative these patterns would be of actual usage. However, we believe that both RCS and GSS describe perfectly plausible applications, while OSS is sufficiently generic and non-specific to allow a multitude of different variant configurations to be generated. Other advantage of resorting to automated test case generation include the ability to use any and all models in the repository for validation and testing purposes (rather than just those for which manual annotation would be a viable option), allowing very large data sets to be used.

Algorithm 7 was used to annotate a given base feature model with soft constraints according to the RCS, GSS and OSS patterns. The density parameters  $D_{RCS}$ ,  $D_{GSS}$ , and  $D_{OSS}$  control the number of soft constraints introduced. Nevertheless, fluctuations may occur because duplicates may be generated, while redundant soft constraints and obviously non-sensical scenarios such as the one in Figure 4.9 are also ignored (lines 23 and 24, respectively). This makes it possible that the actual number of soft constraints injected into the feature model is less than the maximum  $N_c * D_{RCS} + N_G * D_{GSS} + (N_O + N_{fG}) * D_{OSS}$ , where  $N_{fG}$  is the number of group children features,  $N_G$  the number of groups,  $N_O$  the number of optional features and  $N_C$  the number of hard constraints. This approach is effective and simpler than trying to always generate valid, distinct soft constraints in sufficient number (which may very well be impossible, depending on the chosen density parameters and structural properties of the base feature model  $F$ ).

To create the test data set so that a large number of soft constraints, distributed across multiple feature models, is obtained, we applied Algorithm 7 to all the feature models in Table 7.1, with density parameters  $D_{RCS}$ ,  $D_{OSS}$ , and  $D_{GSS}$  all set to 1. This ensures that the injection algorithm will try to inject one *RCS* per each constraint, one *OSS* per optional and group children, and one *GSS* per group, thereby achieving a high density of soft constraints in each feature model. If an experiment requires lower variant densities, a smaller subset of soft constraints may be selected from among all those that are available.

The full results of the injection process can be found in Table C.1. An overview

---

**Algorithm 7:** Soft constraint injection algorithm
 

---

```

1 Function injectSoftConstraints
   input : Feature model  $F$ , density parameters  $D_{RCS}, D_{GSS}, D_{OSS}$ 
   output: Annotated feature model

2    $N_C \leftarrow$  number of constraints in  $F$ 
3    $N_G \leftarrow$  number of groups in  $F$ 
4    $N_O \leftarrow$  number of optional features in  $F$ 
5    $i = 0$ 
6   while  $i < N_C * D_{RCS}$  do
7      $i \leftarrow i + 1$ 
8     select random constraint  $c$  from  $F$ 
9     inject reverseConstraintSuggestion ( $c$ ) into  $F$ 
10   $i = 0$ 
11  while  $i < N_G * D_{GSS}$  do
12     $i \leftarrow i + 1$ 
13    select random group  $g$  from  $F$ 
14    inject groupSelectionSuggestion ( $g$ ) into  $F$ 
15   $S \leftarrow$  optional features  $\cup$  group children of  $F$ 
16   $s \leftarrow S.size()$ 
17   $i \leftarrow 0$ 
18  if  $s > 1$  then
19    while  $i < s * D_{OSS}$  do
20       $i \leftarrow i + 1$ 
21       $o_1 \leftarrow$  random feature from  $S$ 
22       $o_2 \leftarrow$  random feature  $o_2$  from  $S$  such that  $o_1 \neq o_2$ 
23      if  $\neg o_2.isAncestorOf(o_1)$  AND
24         $\neg childrenOfSameAlternativeGroup(o_1, o_2)$  then
25        [ inject optionalSelectionSuggestion ( $o_1, o_2$ ) into  $F$ 
26  return  $F$ 

```

---

is presented in Table 7.2. A total of 1492 soft constraints were generated and injected across all feature models, with an average number of 38.26 injected constraints per model.

Table 7.2: Injection results

Injected constraints	Quantity
Total number	1492
per model	38.26

### 7.1.3 Unsatisfiable and Untriggerable Soft Constraint Identification Experiment

We have conducted an experiment to analyze the procedures for identifying unsatisfiable and untriggerable soft constraint (Equations (4.1) and (4.5) in Sections 4.3.2.2 and 4.3.2.3, respectively). These two procedures are tested in the same experiment because, unlike contradictory soft constraints, unsatisfiable and untriggerable soft constraints can be identified by independent examination of each soft constraint (while for contradictions, a set of constraints must be always involved, requiring a different experimental procedure).

#### 7.1.3.1 Results

Equations (4.1) and (4.5) were applied and we obtained the results in Table C.2 of the Appendix, summarized here in Table 7.3. Table 7.3a displays the time required for conducting the analysis. Total time was 5.4 [s], the average time per constraint was approximately 3.5 [ms], while the average time per feature model was 139 [ms]. Finally, Table 7.3b displays the result of the identification process: 20 soft constraints were untriggerable but none were unsatisfiable.

Figure 7.1 presents a box and whiskers plot of the execution time per constraint. The interquartile range (IQR), that is, the difference between the third and first quartile is 1.52 [ms], which means that results are densely packed around the median value of 1.55 [ms]. Seven outliers can be found in the upper range. These outliers can be easily explained:

- The close proximity of the median with the absolute minimum value of 0 [ms] does not allow dispersion towards the left (lower) side of the distribution.

- External sources of measurement noise such as operating system activity can have a significant impact due to the diminutive intervals of time being considered here.

Table 7.3: Aggregated results for unsatisfiable and untriggerable soft constraint identification

Running time	
Total	5.4 [s]
per constraint	3.49 [ms]
per model	139.05 [ms]

(a) Performance results

Constraint type	Nr. Found
Unsatisfiable	0
Untriggerable	20

(b) Identification results

Corresponding results presenting the distribution of the analysis time per feature model can be found in Figure 7.2. The *IQR* is 123.5 [ms], with median 41 [ms]. Results are densely concentrated on the lower values: half of the feature models, corresponding to the first and second quartile, are analysed under 41 [ms], while the next 25% are done in under 151.5 [ms]. Most outliers can be explained by virtue of being feature models with a high number of soft constraints number.

### 7.1.4 Contradictory Soft Constraint Identification Experiment

In this section we describe an experiment we have conducted to analyze the procedure for identifying contradictory constraints (discussed in Section 4.3.2.2). Contradictions always involve at least two constraints.

#### 7.1.4.1 Results

Equation (4.2) was applied to identify contradictions between every pair of soft constraints of each feature model in our data set. Aggregated results are presented in this section, while full results can be found in the appendices in Table C.3. These results highlight the performance of the algorithm and help addressing research question 1(a)i.

Another objective of our test is identifying the prevalence of suspicious interactions such as contradictory soft constraints, in accordance to research question 1(a)ii. Figure 7.3 demonstrates the percentage of pairs of soft constraints that

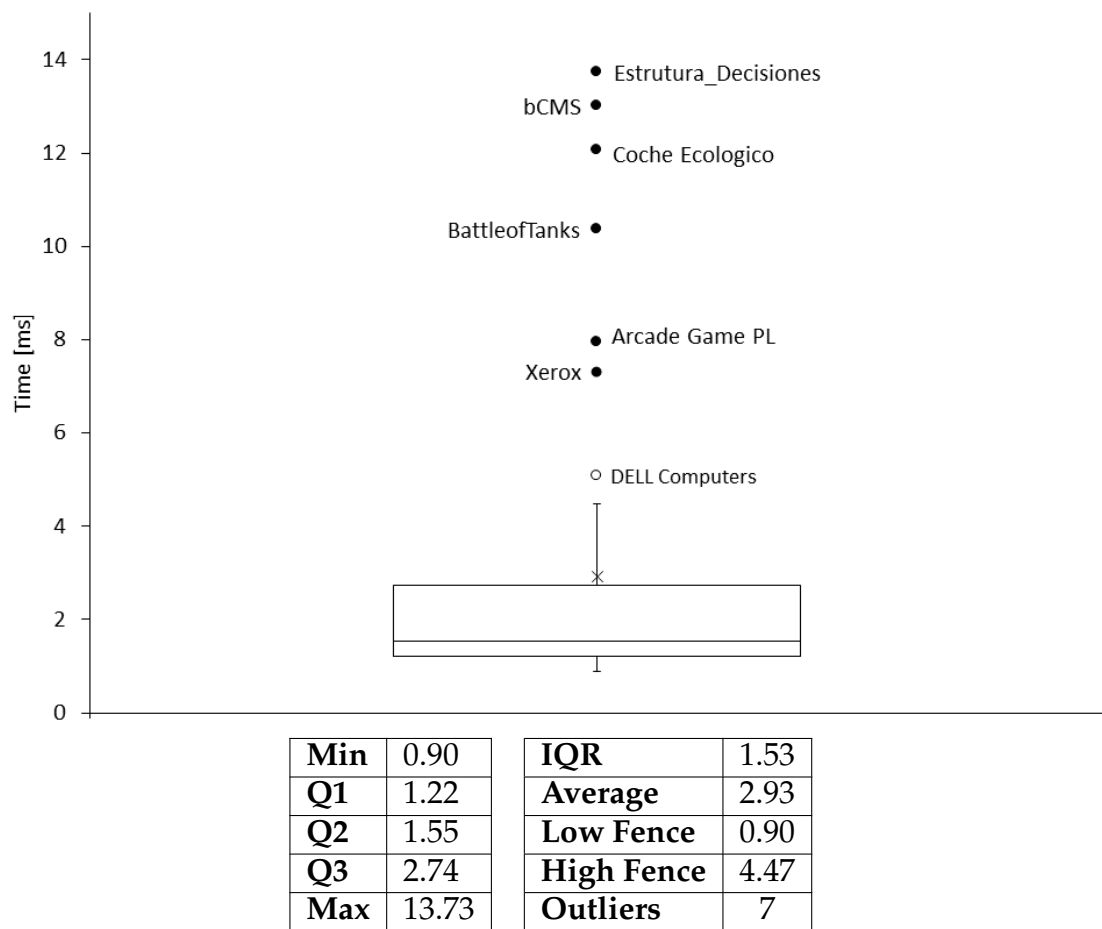
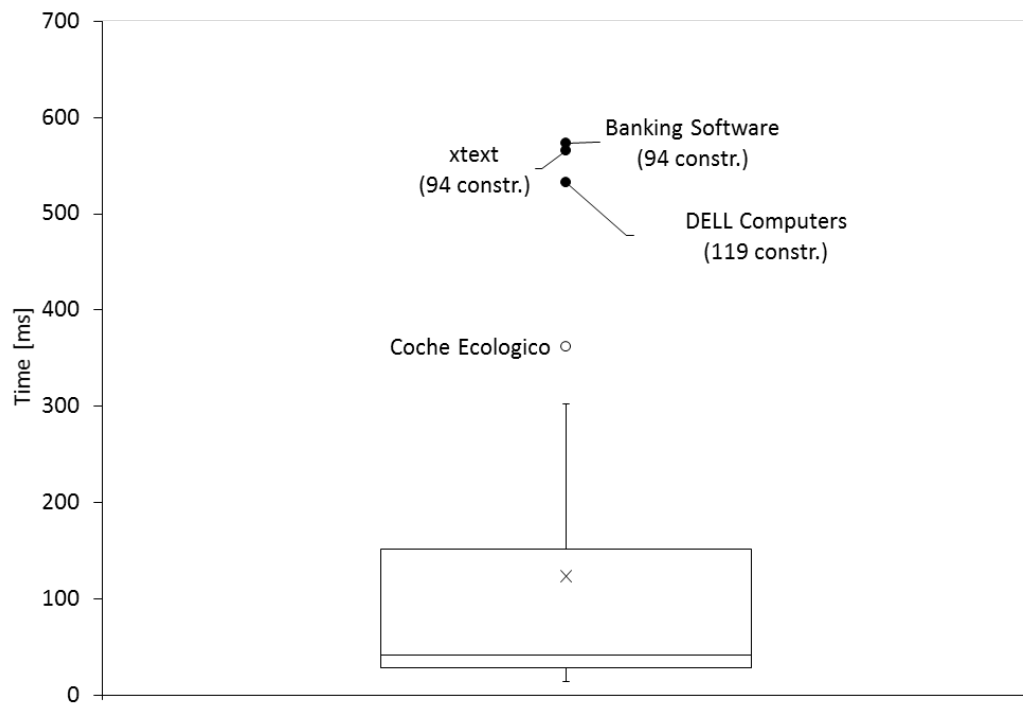


Figure 7.1: Analysis time per constraint



<b>Min</b>	14	<b>IQR</b>	123.5
<b>Q1</b>	28	<b>Average</b>	123.3
<b>Q2</b>	41	<b>Low Fence</b>	14
<b>Q3</b>	151.5	<b>High Fence</b>	302.0
<b>Max</b>	573	<b>Outliers</b>	4

Figure 7.2: Analysis time per feature model

were found to be contradictory. In the central half of the models (second and third quartile), this value ranged from 4.2% up to 12.8%. The three outliers are the feature models with higher clause density.

Concerning the performance of the identification algorithm, we find that it makes sense to relate performance to the intended application of our algorithm: to be used in real time during feature model edition. In this way, the identification procedure would be run to identify contradictions whenever a new soft constraint was added to the model. This would have no impact on pre-existing contradictions involving other constraints, so all that is required in this case is to identify contradictions in which the newly added constraint is involved. Therefore, in Figure 7.4 we present the time required for identifying all contradictions involving one specific constraint. This value was computed for all constraints in each model, and the results represent the average values. It can be seen that results for the second and third quartile ranges from 7.26 to 26.10 [ms], indicating very good performance on the average case. The highest outlier takes on average 336 [ms] per constraint, which is still compatible with real-time identification (since the identification algorithm only runs once for each newly introduced soft constraint). The main factor explaining the outliers is the high number of constraints (which increases the total number of pairs that must be analyzed).

## 7.2 Configuration Repair Testing

We have conducted a series of tests to assess the performance of our configuration repair technique. These tests used the models available in the S.P.L.O.T. repository at the time of access (Feb. 2014) with over 40 features. These tests have the purpose of assessing the performance of the partitioning, cover computation and repair mechanisms.

### 7.2.1 Experiment Objectives and Goals

The configuration repair algorithm is at the core of the prototype-based configuration approach, and this experiment relates to research questions 2, 3, 4, and 5. This has the purpose of:

- **Assessing the performance of the repair mechanism.** The repair algorithm is divided into two distinct steps: an initial computation of the cover of the feature model (executed once per feature model), and then the identification of the repairs for a given configuration (executed once per invalid

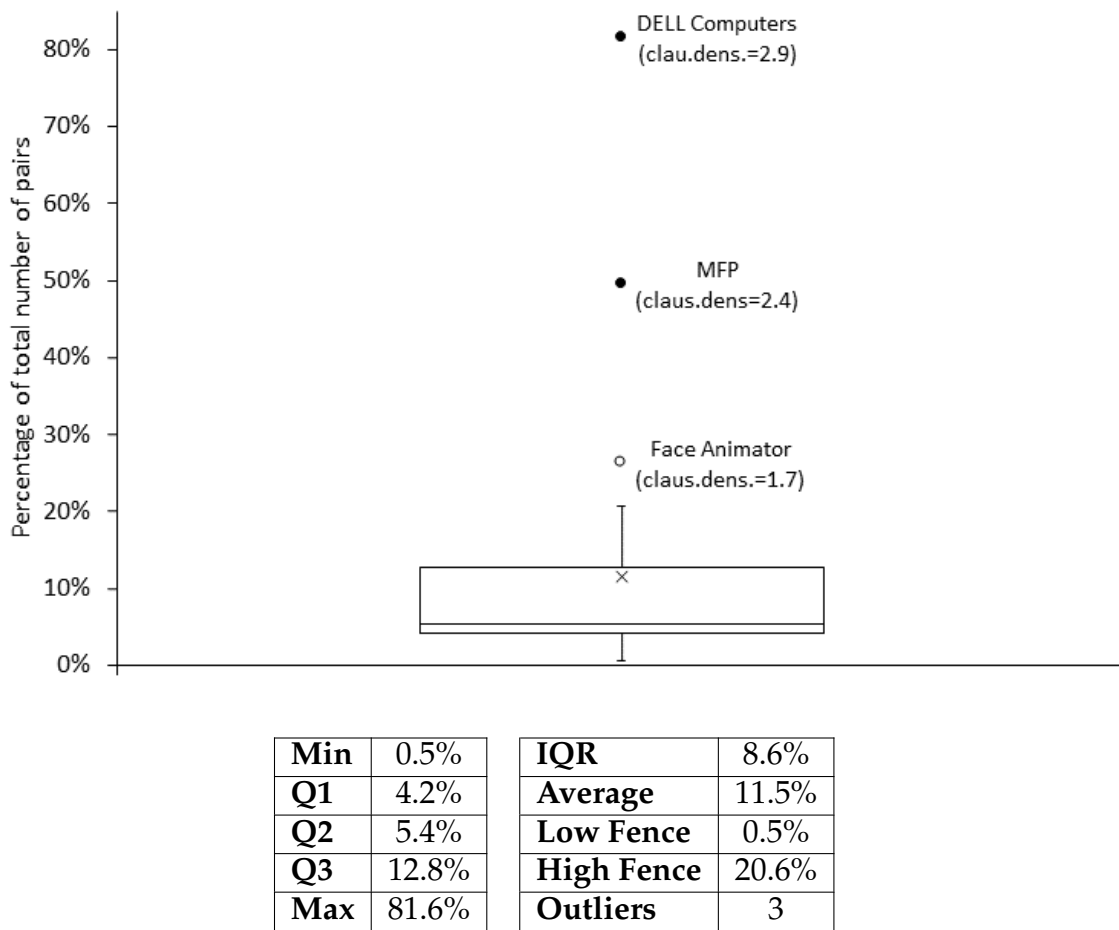


Figure 7.3: Percentage of contradictory pairs

configuration). In this way, the performance of the repair mechanism can be divided into two different aspects:

- *Performance of Partitioning and Cover Computation.* This is the most computationally expensive operation, but it must be performed only once per feature model.
- *Performance of Repair Identification.* This operation must be performed only once per repair operation, and should be highly efficient.

The configuration repair algorithm is a crucial step of the prototype-based approach. Therefore, investigating its performance is relevant for addressing research question 5.

- **Assessing the effectiveness of the problem-decomposition approach.** The problem-decomposition approach aims to reduce the number of alternative choices the user is faced with when deciding how to repair an invalid options. It is therefore relevant for research question 3b.

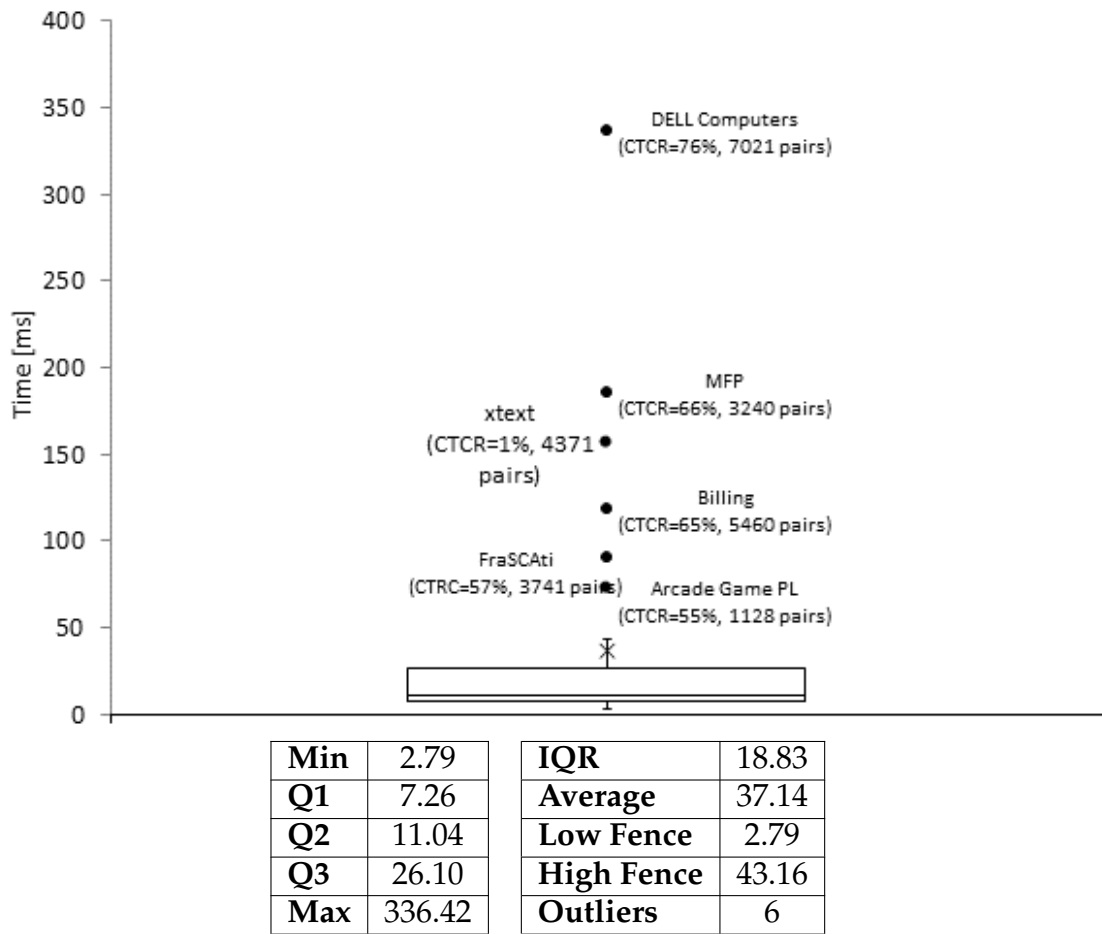


Figure 7.4: Analysis time for all contradictions with one specific constraint [ms]

Performance of the cover computation step, and also the effectiveness of the problem-decomposition approach, depends on the effectiveness of the partitioning. Highly constrained models are less amenable to partitioning, so the extent to which this is (or isn't) a relevant factor in the type of models used in our tests must be assessed.

Repair identification is expected to be a highly efficient operation. Even if performance of the partitioning and cover computation step is found to be poor, this may be compensated by very high repair identification performance, since the high initial overhead will be amortized over each ensuing repair on that feature model.

## 7.2.2 Partitioning and Cover computation

The partitioning and cover extraction algorithm was applied to all the feature models included in the test. After an initial partitioning step, described in Section 6.3.2, the cover for each partition is obtained by using the state of the art

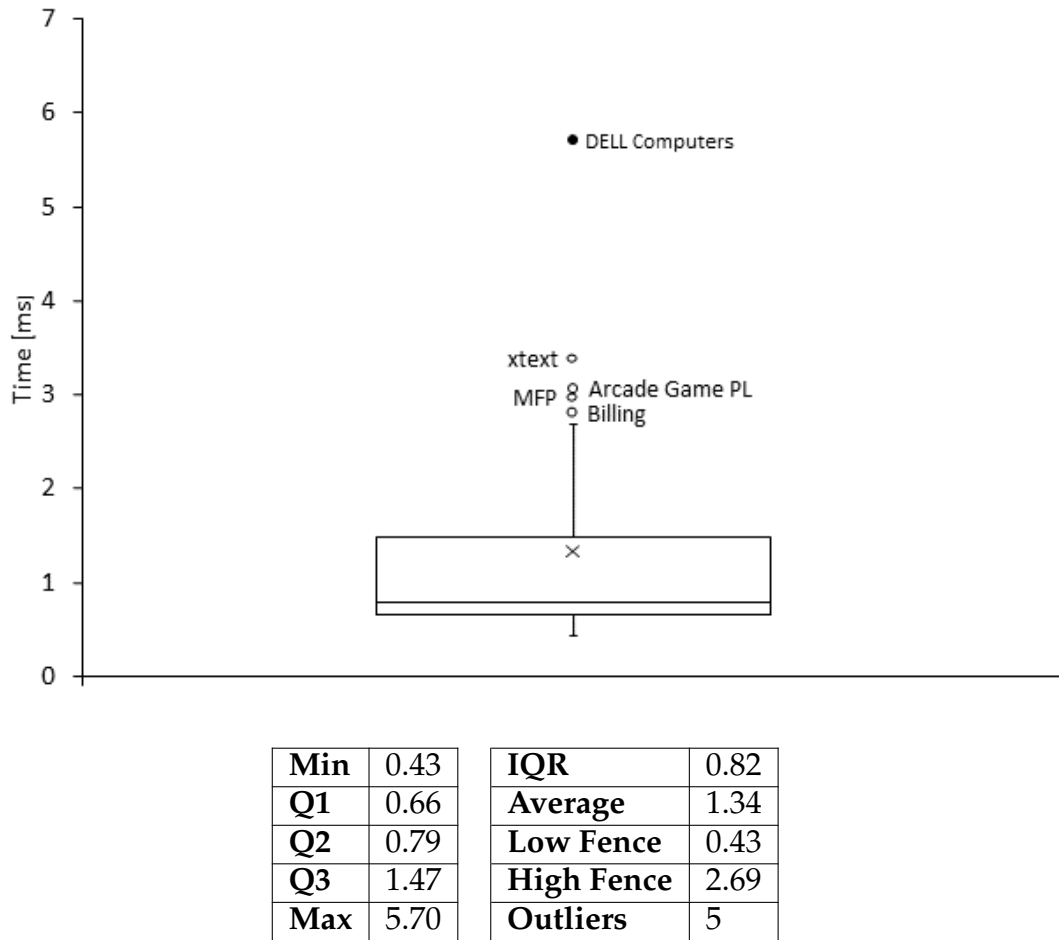


Figure 7.5: Analysis time of a pair of constraints [ms]

Espresso heuristic logic minimizer [Rud86]. These are the initial steps of the configuration repair process, involving a fairly complex operation (cover extraction), that however must be performed only once per feature model.

### 7.2.2.1 Results

Full results for the partitioning and cover extraction processes are described in Table C.4 of the appendices. An extract from these results, including the largest feature models, is presented here in Table 7.4. The "Partitions" column shows the number of different partitions into which the model is broken down. The next column displays the total "Number of terms" ( $T_P$ ) included in the cover description<sup>3</sup>. The "Partitioned Terms" ( $T_P$ ) column contains the total number of terms of the covers of the the partitions. The ratio  $T_P/T_F$  is important for assessing the

<sup>3</sup>A cover is a disjunction of *terms*. Each term is a conjunction of one or more variables (or their negation).

extent by which the number of terms to be considered is reduced by the partitioning process. The rightmost column displays the total time required to compute the partition and the cover. Results are indicative of the excellent performance of the algorithm in the overwhelming majority of the cases. Even models whose cover includes billions of terms can be successfully analyzed in a few hundred milliseconds, thanks to the partitioning technique. Considering that cover minimization is the more computationally expensive operation and that it needs to be performed only once per feature model (and not once per repair), these results indicate that our approach is well suited for handling the vast majority of models available in the public repository. The only instance where our approach was not successful in obtaining a minimal cover within reasonable time (1 hour) was with the "Decision Making" model. The partitioning algorithm was not capable of significantly breaking down this specific model. This is because the model is highly constrained with significant coupling between different branches of the feature tree. This factor, combined with the high number of optional features, makes it extremely challenging to successfully break down, process and analyze this model.

Table 7.4: Extract of the results for largest models

Model Name	Partitions	Number of Terms ( $T_F$ )	Partitioned Terms ( $T_P$ )	$T_P/T_F$	Time [ms]
Decision Making	3	-	-	-	-
BankingSoftware	66	32060448	321	0.00%	1747
TankWar	14	42996610800	130	0.00%	392
Billing	19	24	42	175.00%	579
Coche_ecologico	44	207360	75	0.04%	1357
xtext	25	5231304	127	0.00%	664
Xerox	49	8707129344000	144	0.00%	1404
Model_Transfor.	22	207152640	165	0.00%	599
J2EE web architecture	29	5225472	57	0.00%	901

The effectiveness of the partitioning process in reducing the number of terms is reflected on the ratio  $T_P/T_F$ . Since the algorithm running time is dependent on the number of terms, this ratio offers a clear indication of the benefits of the partitioning approach to repair performance. Figure 7.6 presents the aggregated results. Outliers are not represented for graphical convenience, but are discussed in the text. It can be seen that in 75% of the cases, the number of terms is reduced to 1.8% or below, while in 50% of the cases a reduction below 0.1% is observed. These results account for an extremely significant improvement of the repair algorithm performance.

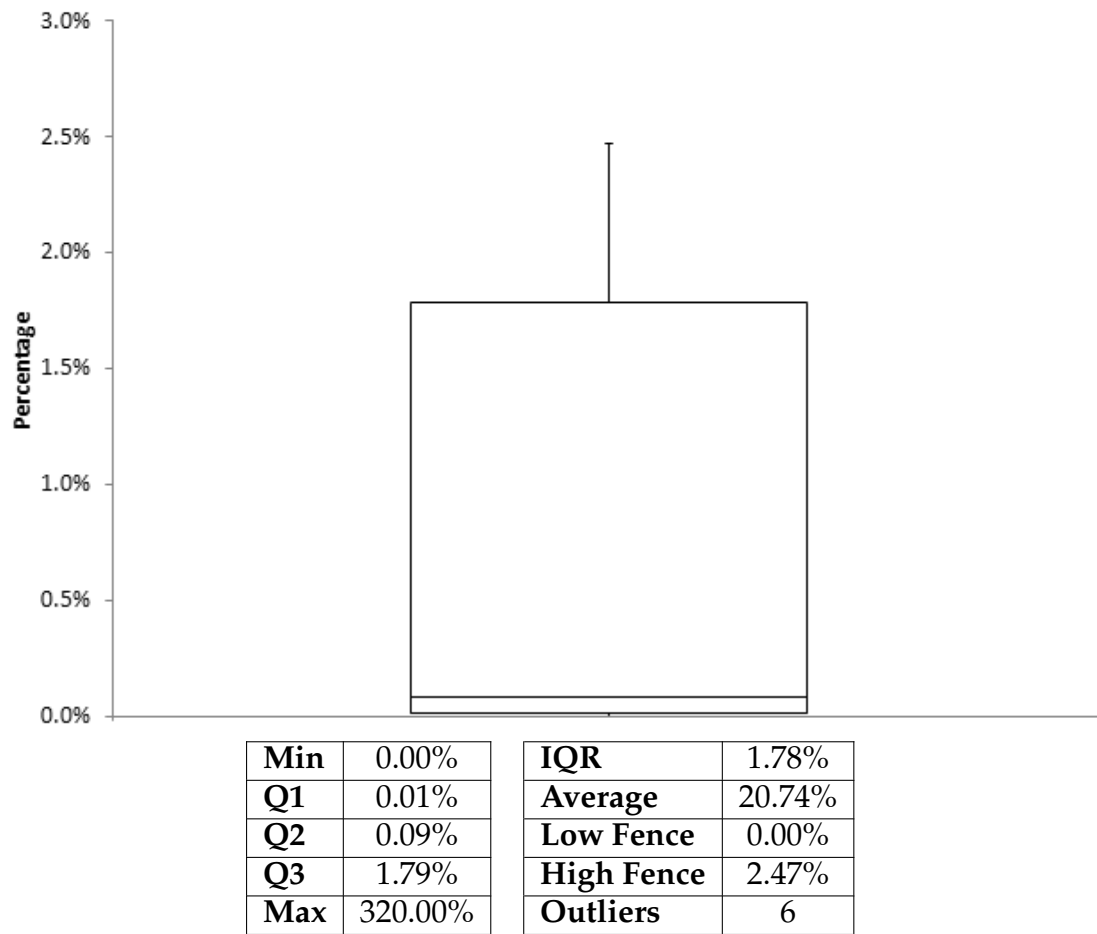


Figure 7.6: Ratio between the number of terms after partitioning and before partitioning ( $T_P/T_F$ ) (outliers not represented)

Six outliers, found in Table 7.5, were observed. Without exception, all these cases have comparatively low number of terms, making term compression less relevant. Reduction in the number of terms was nevertheless achieved in all but three cases. In two of those, the number of terms was left essentially unchanged. This was mainly due to the highly constrained nature of these two cases, effectively preventing partitioning in one case (DELL COMPUTERS) and severely impairing it in the other (Arcade Game). In the latter case, although the model is partitioned into 16 submodels, partitioning is not balanced: most of the original model is contained in a single submodel. In the MFP test case, a three-fold increase was actually observed, due to the number of partitions being superior to the number of terms. However, the number of terms itself is residual making the 320% increase largely irrelevant.

Table 7.5: Partitioning - Outlier cases

Model Name	Partitions	Terms	Partitioned Terms	$T_F/T_P$	Time [ms]
Documentation_Generation	39	560	53	9.5%	927
thread	5	15162	2534	16.7%	586
FraSCAti	14	1760	455	25.9%	396
MFP	12	5	16	320.0%	418
Arcade_Game	16	5418	5433	100.3%	3652
Dell_Computers	1	853	853	100.0%	471

Concerning the runtime performance of the partitioning algorithm, Figure 7.7 demonstrates that for the large majority of test cases, it takes less than 1 second for the partitioning and cover extraction algorithm to execute. This is very significant, as partitioning and cover computation is the most time intensive step of the algorithm (that must be performed only once per feature model). The only outlier that meaningfully deviates is "Arcade Game". This is most likely due to lower partitioning performance (see Table 7.5) resulting in higher number of partitioned terms, making cover extraction a lengthier process.

### 7.2.3 Repairing Random Invalid Configurations

We conducted a series of experiments targeting the repair of random invalid configurations. These configurations were obtained by iteratively creating random configurations until an invalid solution was found. Solutions were generated with a distribution of 45/45/10 percent for selected/deselected/open features, respectively. We generated 30 such configurations for each model.

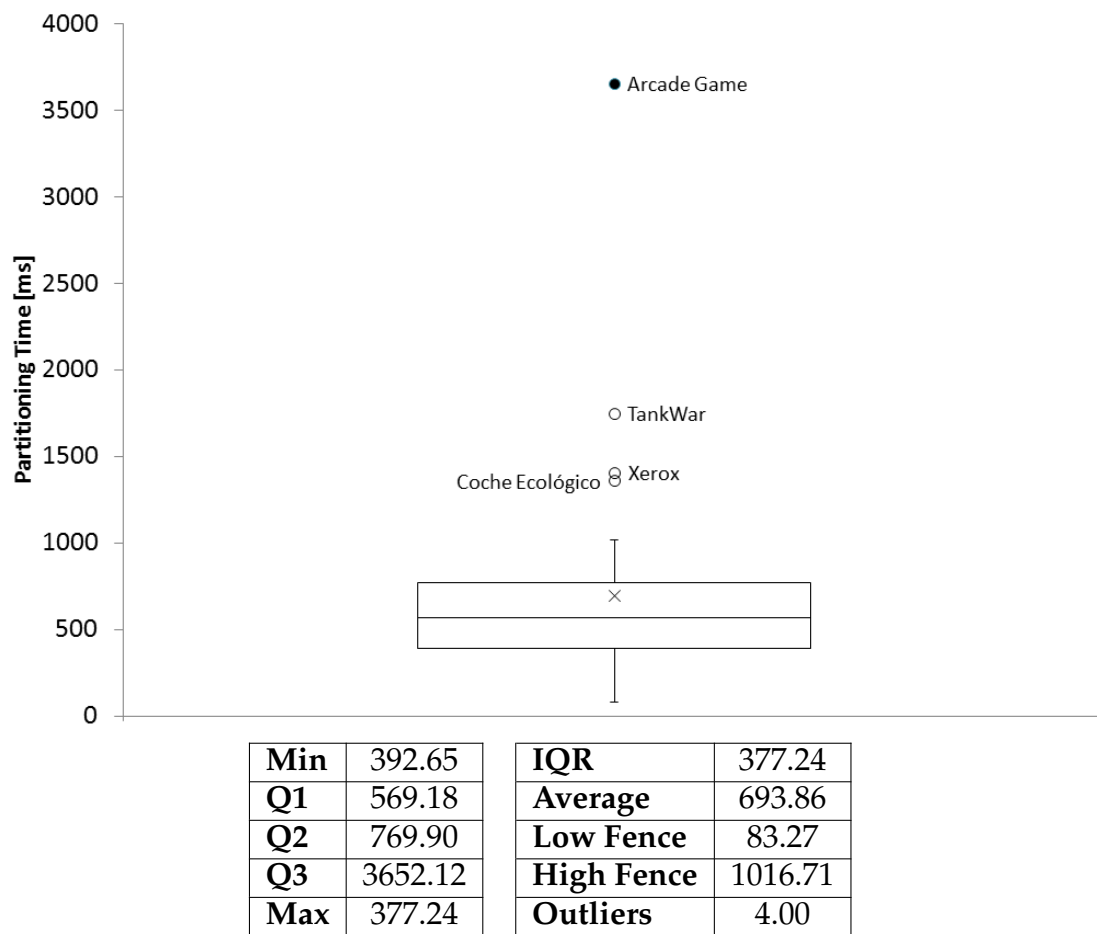


Figure 7.7: Time required for execution of partitioning algorithm

### 7.2.3.1 Results

Results for an optimization criterion minimizing the Hamming distance between the original and repaired configuration are presented in Table C.5 of the appendices. An excerpt of the results is presented in Table 7.6.

Analysis of the performance of the algorithm is separated into two factors: the time to identify the first solution (i.e., the time required for building the repair iterator) and the time to iterate over the remaining solutions. Since the second factor is dependent on the number of alternative repairs identified, we present in the fourth column the time required to iterate over all identified solutions, but only up to a maximum number of 1000 solutions. It can be seen that the repair algorithm is extremely efficient, not only in identifying the first potential repair, but also in iterating over other admissible solutions.

Table 7.6: Repair of random configurations — excerpt of full results

Model Name	Average Number of Repairs	Time to Find First Repair (ms)	Iteration over next repairs (max. 1k) (ms)
BankingSoftware	43.9	3.2	0.2
TankWar	371819.6	0.5	9.5
Billing	2.3	0.6	8.5
Xerox	7166361626.9	0.4	1570.3
Arcade_Game	1.7	28.5	0.3
Coche_ecologico	75.5	0.2	1.2
xtext	26.8	0.5	0.1
Model_Transformation	104.1	0.3	1.4
J2EE web architecture	55.4	0.1	3.7

The time taken to find the first repair was 28.54 [ms] on the worst case (Arcade Game PL). For the significant majority of cases, less than 1 [ms] is required, as can be seen in Figure 7.8. This confirms excellent repair identification performance of our approach.

To explore and demonstrate the behavior of the algorithm with alternative minimization criteria, we created an evaluator that promotes the preservation of selected features in the repaired solution. In this case, a repair is evaluated by the formula in Equation 7.1

$$D = T_{ONtoOFF} * K + T \quad (7.1)$$

Where  $D$  is the "distance" to be minimized,  $T_{ONtoOFF}$  the number of features selected in the original configuration that are toggled, from selected to deselected,

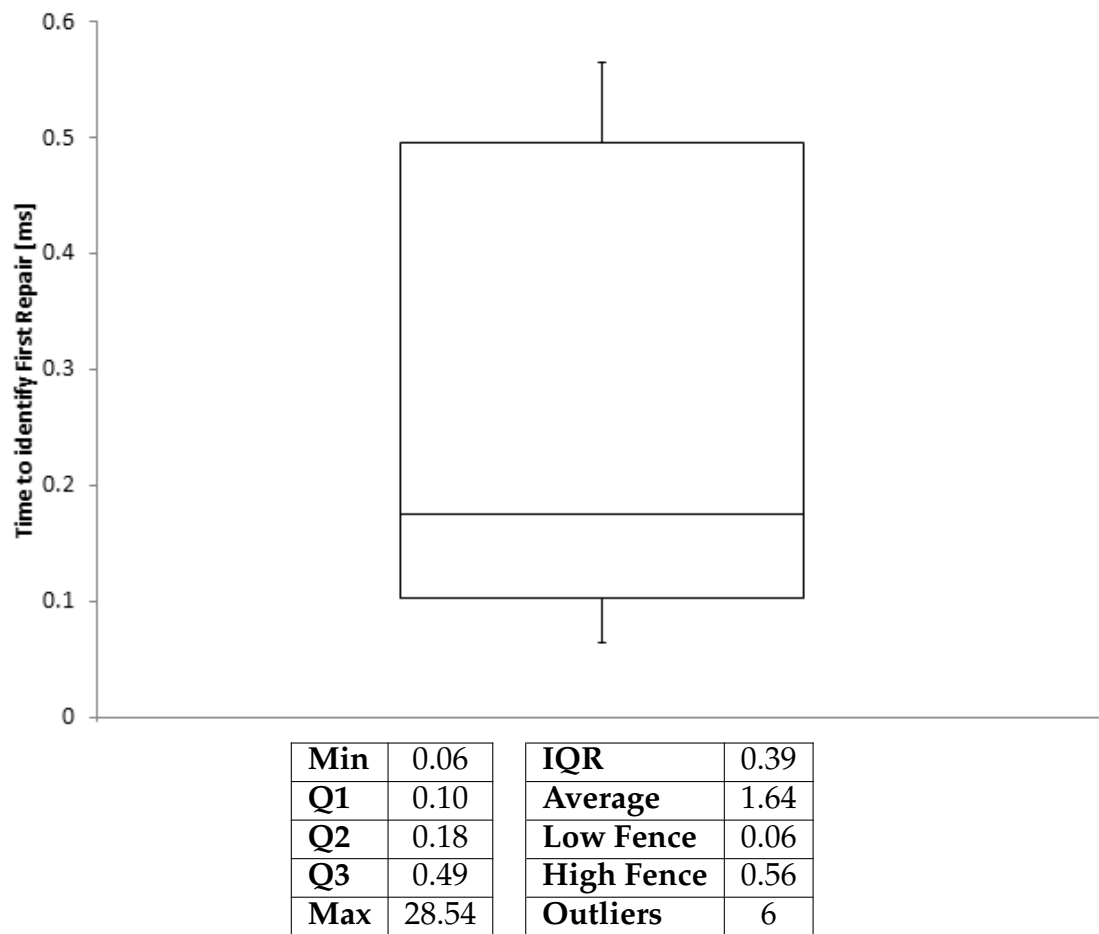


Figure 7.8: Time to find first repair (ms). Outliers not represented.

in the repaired configuration,  $K$  a large number (greater than the total number of features of the model) and  $T$  the total number of features toggled from the original to the repaired configuration. This metric ensures that preferred repairs will be those that preserve the most features originally selected in the configuration, with secondary optimization of Hamming distance. Results can be found in Table C.6 in the appendices, and a short extract can be found here in Table 7.7. It can be observed that performance is not affected by the alternative nature of the evaluator and that results have been modulated by the introduction of a new selection criteria.

Table 7.7: Extract from repair results — Finding repairs for invalid configurations that preserve selected features

Model Name	Average Number of Repairs	Time First [ms]	Iteration over next (max. 1k) [ms]
BankingSoftware	65.33	0.81	0.02
BattleofTanks	647985.20	0.27	5.49
Billing	1.50	0.23	0.05
Coche_ecologico	28.50	0.33	0.09
xtext	1.00	0.40	0.02
Xerox	20.77	0.33	60.58
Model_Transformation	34.90	0.24	0.58
J2EE web architecture	3.33	0.15	0.05

The single repair identified for the xtext model can be understood by virtue of the specific characteristics of this model. It is poorly constrained, and also does not include any groups. Therefore, invalid solutions arise because of violations of the feature tree structure only: children features are selected while their parent is not. Possible solutions for such a problem could involve, if a Hamming metric was being used, either selection of the parent or deselection of the child, potentially allowing for multiple optimal solutions to be found. However, the new metric used in this scenario gives preference to resolutions that preserve selected features. In this way, the optimal solution is always the first one: selecting the parent of selected features. This gives rise to the single solution shown in Table 7.7, and further demonstrates that the new optimization metric was indeed successfully handled by the repair algorithm.

## 7.2.4 Problem Decomposition

Results in Section 7.2.3 (and Table C.5) have highlighted the prevalence of alternative repair options in the overwhelming majority of scenarios and models. While in some cases the number of alternative repair options is reasonably small, making exhaustive listing a viable option, in the generic case this is not true. While hard limits can be difficult to define, it can be stated without a doubt that instances where the number of alternative repairs is in the order of thousands or even millions cannot be handled reasonably by exhaustive listing. In practice, lists can become cumbersome with a much lower element count. We intend to verify the capability of our approach to reduce the number of alternative repair options presented to the user.

### 7.2.4.1 Results

The purpose of problem decomposition is reducing the number of repair options presented to the user down to a manageable size. Therefore, for relevance, we address only results for repairs of random configurations for which the average number of repairs is greater than 50. Results can be found in Table 7.8, in which

Table 7.8: Problem decomposition of repairs of random configurations.

Model Name	Total Repairs	Problems	$\frac{\text{Repairs}}{\text{Problem}}$	Compression
Dell_Computers	562	1	562.0	0.0
Coche_ecologico	4536	7	4.0	99.4
Meeting_Config	108	4	3.3	88.0
Eclipse1-Reuso	128	4	4.0	87.5
Car	10368	12	2.3	99.7
ConsolasVideojuegos	144	3	5.7	88.2
AndroidSPL	128	4	4.0	87.5
Xerox	320	3	9.3	91.3
DS Sample	6912	6	5.7	99.5
Letovanje	256	3	12.7	85.2
Electronic_Drum	248832	11	3.2	100.0
Jogo	12288	13	2.1	99.8
Model_Transformation	3888	8	3.3	99.3
his	64	6	2.0	81.3
thread	72	1	72.0	0.0
BankingSoftware	10368	7	5.1	99.7
BattleofTanks	766771200	9	11.0	100.0

we can find the model name, total number of repairs found, number of problems into which the repairs were decomposed and compression rate achieved. The

compression rate represents the reduction in the number of choices provided to the user, and is computed as  $100\% - \frac{\text{problems} \times \text{repairs} / \text{problem}}{\text{total repairs}}$ . Significant levels of compression can be identified in all cases except for the two models that were not decomposed in several partitions by the algorithm described in Section 6.3.2. A significant result is that all the most problematic instances (with thousands and even millions of alternative repairs) were successfully compressed into a set of problems of very modest size.

## 7.3 Empirical Testing of Enhanced Configuration Support

We have conducted an empirical test for assessing configuration assistance based on soft constraints. Test subjects with industrial and academic backgrounds were invited to participate in the experiment, in which they used our prototype tool to configure a feature model. After a short training session (45 minutes), each user was provided with the description of four different products, corresponding to four different feature models, including a set of desired properties to include, if possible, in the created configurations. For each user, soft constraint-based support was provided for creating two out of the four required configurations, setting up in this way a baseline for contrasting assisted vs non-assisted configuration. After creating the configuration, the test subjects were asked to fill an online survey.

This experiment provides important information that is relevant for research questions 4 and 5.

### 7.3.1 Experiment Design

We began the design of this experiment by creating a Goal/Question/Metric document [BCR94] (See Section D in the appendices). This helped defining the purpose of the experiment and the metrics that were to be measured. In synthesis, we aim to measure the effectiveness and efficiency improvements due to the adoption of soft-constraint configuration support, using quantitative and qualitative feedback obtained from the experiment participants. To achieve the objectives of the experiment, we decided that a within-subject approach (a.k.a. repeated measures or within groups design) would be the preferable choice, rather than resorting a between-subjects approach with control group. In a within-subject

approach, instead of splitting the test population into groups subjected to different treatment or tests, each individual is subjected to all variant treatments. In the case of our test, this entails that each participant will be asked to complete the four test cases, having support for only two of them.

The within-subject approach has the following advantages:

- **Does not require a large pool of participants.** Anticipating the difficulty of finding a large number of volunteers that qualify for participation, this is a significant advantage.
- **Helps addressing population heterogeneity.** Since all individuals are exposed to the same tests, this helps addressing the natural variation of participant capabilities and skills.

Major drawbacks of this approach are carryover effects from early tests. These include fatigue (artificially driving down performance in later tests) and conversely learning effects (practice gained in earlier tests can improve later performance). To address these, we applied a counterbalancing design, where the order by which different participants take the successive tests is changed. This ensures that carryover effects are more evenly distributed throughout the results. To find the order by which each participant conducts the required tests, we considered all the possible 24 permutations of test order. Each such permutation describes the order by which one participant will conduct the test. If the number of participants is a 24 (or a multiple of 24), then all possible orderings are contemplated. However, if the number of participants is less than 24, then some permutations will be left untested. One potential problem with this is that permutations are naturally generated in an order for which variation of the rightmost elements is higher than variation of the leftmost elements (e.g., ABCD, ABDC, ACBD, ACDB, ...). To avoid negative impacts in the results due to this effect, when less than 24 participants are involved, permutations are shuffled to a random order before being assigned to test participants. This achieves a more uniform distribution of the test order.

Four test cases were created based on feature models from the S.P.L.O.T. online repository [MBC09]. We selected four feature models of medium to medium-high dimension, in domains that should be at least somewhat familiar for software and computer engineering professionals and academics (see Table 7.9). These test scenarios include a series of costumer constraints representing the client requirements and desires for a specific product to be created. The test cases also include a brief textual overview of the feature model. The test cases can be found

in Appendix E. The feature model itself was not reproduced in the test case descriptions, as it could be observed and interacted via the prototype tool during the experiment.

Table 7.9: Feature models selected for test cases

Model Name	Features	Optional	Groups	Hard Constr.	CTRC	Clause Density
DELL Computers	46	1	8	110	76%	2.9
Experiment Environ.	35	0	6	20	80%	0.7
OW2-FraSCAti-1.4	63	39	2	46	57%	1.3
Web_Portal	43	17	6	6	25%	0.5

A 45 minute presentation was created where the concepts of feature modeling and standard and enhanced configuration support were presented. The various options of our prototype tool were also demonstrated, and the participants were offered the opportunity of experimenting hands-on with a test scenario. The prototype tool was fully automated to ensure that navigation and progression through the training and test cases was done automatically and in the correct order depending on the identification numbers assigned to each participant. Results were also automatically collected and submitted with no user intervention, so that human error during data collection and experimental set-up is minimized and disruption or distraction from testing activities is minimized.

A few test trials were conducted to assess duration of the experiment, phrasing and clarity of the questions and case descriptions and suitability of the training material. This initial feedback resulted in minor adjustments before the first experiments were conducted.

### 7.3.2 Data Analysis and Hypothesis

Some participants are generically faster/more efficient than others, and the number of constraints and dimension is not the same for all the test cases. To address this heterogeneity, we focus our analysis on the relative performance and improvement that each participant experienced. Therefore, we normalized results in each test case to the 0-100% scale, where 0% and 100% correspond to the minimum and maximum observed value in all participants for that test case. This normalization is applied to both configuration time and number of satisfied constraints. The overall performance of each participant, with and without enhanced configuration support, is then measured as the sum of its normalized results in the

individual tests. Therefore, for each test case  $t$ , the performance of each individual participant  $p$  is given by

$$S_{T,p,t} = \frac{T_{p,t} - \min_x T_{x,t}}{\max_x T_{x,t} - \min_x T_{x,t}} \quad (7.2)$$

and

$$S_{C,p,t} = \frac{C_{p,t} - \min_x C_{x,t}}{\max_x C_{x,t} - \min_x C_{x,t}} \quad (7.3)$$

where  $S_{T,p,t}$  is the normalized time performance of participant  $p$  for test case  $t$ ,  $T_{p,t}$  is the time taken by participant  $p$  to complete the test case  $t$ , and  $\min_x T_{x,t}$  and  $\max_x T_{x,t}$  represent respectively the minimum and maximum times of completion, measured over all participants.  $S_{C,p,t}$  is, correspondingly, the normalized constraint satisfaction value,  $C_{p,t}$  the total number of constraints satisfied in test  $t$  by participant  $p$ , while the minimum and maximum values are computed identically to 7.2, *mutatis mutandi*. The performance of each individual can then be aggregated into 4 statistics:

$$P_{T,p}^{\text{WO}} = \sum_{i \notin \text{ENH}_p} S_{T,p,i} \quad (7.4)$$

$$P_{T,p}^{\text{ENH}} = \sum_{i \in \text{ENH}_p} S_{T,p,i} \quad (7.5)$$

$$P_{C,p}^{\text{WO}} = \sum_{i \notin \text{ENH}_p} S_{C,p,i} \quad (7.6)$$

$$P_{C,p}^{\text{ENH}} = \sum_{i \in \text{ENH}_p} S_{C,p,i}, \quad (7.7)$$

where  $\text{ENH}_p$  is the set of test cases for which participant  $p$  benefited from enhanced configuration support.  $P_{T,p}^{\text{WO}}$  and  $P_{C,p}^{\text{WO}}$  measure the time and constraint satisfaction performance without enhanced support, respectively, while  $P_{T,p}^{\text{ENH}}$  and  $P_{C,p}^{\text{ENH}}$  are the corresponding values with enhanced configuration support provided. Lower values of  $P_{T,p}^{\text{ENH}}$  and  $P_{T,p}^{\text{WO}}$  correspond to faster configuration times, while higher values of  $P_{C,p}^{\text{ENH}}$  and  $P_{C,p}^{\text{WO}}$  correspond to better constraint satisfaction ratios. The following hypothesis are then to be verified:

**Hypothesis 1**

Enhanced configuration support reduces the time required to create the configuration.

$$P_{T,p}^{\text{ENH}} < P_{T,p}^{\text{WO}}$$

**Hypothesis 2**

The constraint satisfaction performance improves when enhanced configuration is provided.

$$P_{C,p}^{\text{ENH}} > P_{C,p}^{\text{WO}}$$

**7.3.3 Experiment Realization**

Senior researchers, Ph.D. students and other research partners and collaborators from two higher education institutions participated in the experiment. In spite of strong academic presence, some of the participants have reported significant professional experience, as can be seen in table 7.9.

Due to scheduling difficulties, it was not possible to conduct all tests in a single session. Three sessions were conducted: one at the FCT/UNL site and two at ISEC/IPC. Participants were asked to bring their own laptops so a variety of hardware platforms and operating systems was used. Overall, 13 people participated and submitted their results. Participants were asked to self-assess their competence in three areas of expertise related to this work, in increasing order of specificity,

- Software Engineering
- Feature Modeling
- Feature Model Configuration

Results of this self-assessment are represented in Figure 7.10. Results demonstrate fairly high self-reported Software Engineering competences, while reported competence is lower for the other topics (see Table 7.10).

**7.3.4 Statistical Analysis of Results**

For the purposes of statistical analysis, the results of two participants were excluded:

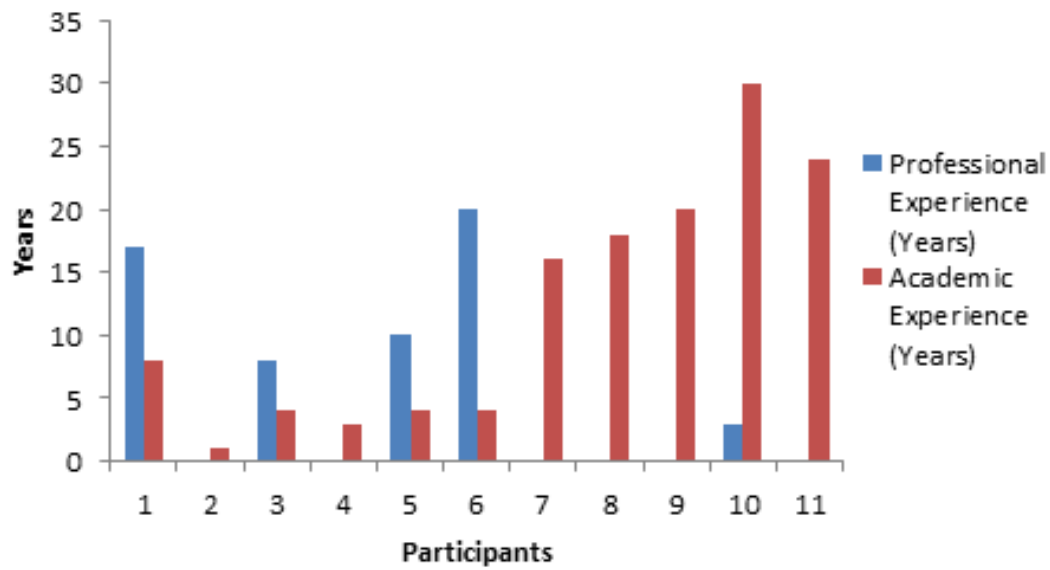


Figure 7.9: Academic and industrial experience of participants (some participants are not represented as they chose not to provide the corresponding optional information).

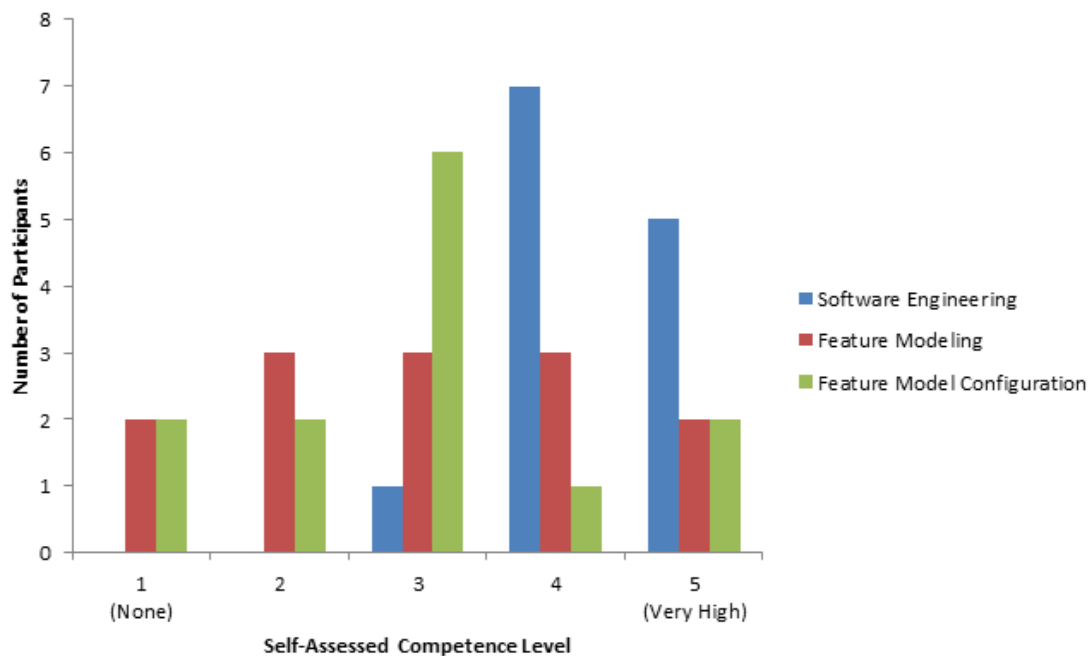


Figure 7.10: Self-assessment of participant competence

Table 7.10: Statistics for self-assessment of participant competence

Skill	Average	95% Confidence Interval	
		Low	High
Soft. Eng.	4.31	3.93	4.69
Feat. Model	3.00	2.18	3.82
Config	2.92	2.16	3.68

- In one case, the participant only realized the availability of one important functionality of the prototype tool (deselecting a feature) halfway through the experience, negating the possibility of establishing an accurate baseline for his performance with respect to later tests.
- In a second case, for reasons that were not disclosed in the feedback form, the participant did not manage or did not wish to complete the configuration in the scenarios without enhanced support.

The test results are analyzed statistically so that the relevant hypothesis (or the corresponding null hypothesis) can be demonstrated. Considering the within-subject approach followed in this experiment, we analyzed the results using the paired samples T-test.

Enhanced configuration support was found to have a statistically significant impact in the effectiveness score from ( $M = 1.26, \sigma = 0.74$ ) to ( $M = 1.92, \sigma = 0.18$ ), with  $t(10) = -2.568, p = 0.028$  ( $M$  is the mean value, while  $\sigma$  is the standard deviation). The significance level  $p$  is below 0.05, so these results indicate that enhanced configuration support does indeed impact favourably the effectiveness of the configuration process: users are able to satisfy more soft constraints when enhanced support is available.

On the other hand, we found that no such improvement could be observed with respect to configuration time. No improvement was observed when changing from standard ( $M = 0.860, \sigma = 0.369$ ) to enhanced configuration support ( $M = 0.868, \sigma = 0.400$ ), with  $t(10) = -0.048$  and  $p = 0.963$ . No significant statistical difference is found between the results, and so no impact of enhanced configuration, positive or negative, can be inferred.

### 7.3.5 User Feedback

#### 7.3.5.1 Perceived benefits and improvements

Participants were asked to indicate their perceived usefulness of enhanced configuration support. To do this, participants were asked three questions for each

case where enhanced support was provided to them:

- Do you feel enhanced configuration support helped you complete your task faster?
- Do you feel it helped you complete your task better (by satisfying more user preferences)?
- Do you feel it helped you understand the required trade-offs, if these existed?

These questions help gauging the perception of participants regarding efficiency and effectiveness gains, as well as comprehension of the required trade-offs. Results are presented in Figure 7.11, while summary statistics can be found in Table 7.11.

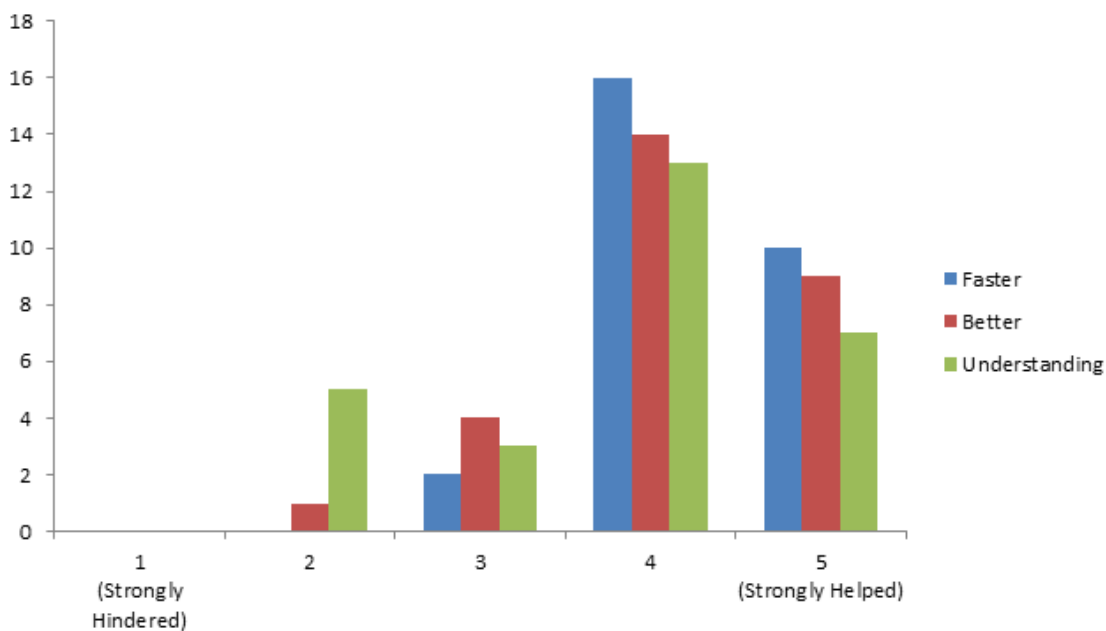


Figure 7.11: Feedback results, describing participant perception of efficiency and effectiveness gains, as well as helpfulness of trade-off information.

Table 7.11: Agregated feedback results

Improvement	Average	Confidence Interval (95%)	
		Low	High
Reducing time	4,29	4,05	4,52
Satisfying constraints	4,11	3,80	4,41
Help understanding trade-offs	3,78	3,39	4,19

It can be seen that results are predominantly positive, with time saving being the most praised improvement, followed in order by effectiveness improvements and trade-off understanding.

#### 7.3.5.2 Observations

Participants were asked to provide additional observations and commentaries. Some comments addressed usability issues such as lack of tooltips, search capabilities or button ordering. These issues can be attributed to the prototype nature of the tool. Although an undo capability was offered, one participant mentioned that more flexible configuration edition capabilities, such as the capability of making pinpoint changes, would have been very useful. Naturally, this is not generically possible due to the effects of user choice propagation, but it does point an interesting area to address in the future.

Specific comments with respect to the core issues in test addressed the trade-off information and suggestion presentation. Some users wished that "more information" was made available so that suggestions and trade-offs could be better understood, although specifics were generally not provided, other than a general desire to have a better understanding of the evolution and closure of the configuration space. Other comments also pointed out that the solution was useful and made it very simple to achieve the stated goals in each case.

One particularly useful comment pointed out that a better understanding of the domain had been achieved in the test cases without enhanced configuration support, simply due to the fact that additional effort had been made to better understand the feature model and its constraints in order to achieve the goals. While this may seem contrary to our purposes, that is in fact not the case: one premise of our approach is to facilitate configuration without requiring deep knowledge of the domain. It stands to reason that in the test cases without support that this knowledge must be more deeply understood to achieve the same or similar results.

Other participants pointed out that even though the configurator suggestions had been ultimately followed, this had not prevented additional exploration of other alternative configurations, due to curiosity or the desire to observe the effects of other choices. This is an interesting point, which may help explaining some of the results.

## 7.4 Results Discussion

### 7.4.1 Identification of Suspicious Interactions

Performance data obtained for the detection of unsatisfiable and untriggerable soft constraints was found to be in the range of a few *ms* only for all the cases (see Figure 7.1). This allows these techniques to be applied in the detection of these potential anomalies in real time, during feature model edition and annotation. This result can be considered according to expectations. Our identification procedures for both contradictions and untriggerable constraints are based on SAT analysis [Coo71]. Even though SAT analysis is known to be an NP-Complete problem [Coo71, GJ90], modern SAT-solvers have been found to be usually effective at handling the Boolean propositions generated by feature model analysis [MAK09]. We found that it is also that case in this particular instance. This can easily be explained, as manipulation of the feature model expression is minimal (conjoining with triggers and/or a single additional constraint), so there would be little reason to believe high efficiency would be lost.

Concerning the prevalence of these type of constraints, it was found to be low. In fact, unsatisfiable constraints were never found in our test scenarios, while untriggerable constraints can be very uncommon. We find that this relatively small prevalence agrees with the exceptional nature of these suspicious modeling scenarios. Considering that constraints are injected according to usage patterns consistent with manual annotation, it would in fact be surprising if we had found that a substantial percentage were potentially anomalous. If soft constraints are included in the feature model through normal Domain Modeling activity (or by a process that attempts to replicate that activity, which is our case), it can be expected that the number of dubious annotations is kept small, as, in practice, modelers can be expected to get things right more often than not. Since unsatisfiability or untriggerability depends on a single soft constraint, it should be relatively simple for the domain engineer to identify and avoid obvious cases of unsatisfiable or untriggerable constraints, such as the ones in Figures 4.7 and 4.9. We explicitly avoid introducing in our tests these types of constraints (e.g., the test in line 24 of Algorithm 7), in an effort to address only annotations that could genuinely arise from sensible manual annotations made by from the domain engineer. This means that any unsatisfiability or untriggerability will likely arise from subtler interactions between the single soft constraint and hard domain constraints, in the context of the feature tree, making them relatively rare occurrences. Conversely,

if completely random and unchecked annotations were used instead, bizarre and unlikely constraints such as *f discourages f* or *g discourages root* (both untriggerable constraints) could easily be generated and artificially drive up the number of anomalies.

These factors certainly contributed in no small way, to the absence of unsatisfiable constraints from the results. Still, our approach only attempts to replicate manual annotation of models, but other scenarios such as product line refactoring or evolution may prove to be more conducive to the appearance of unsatisfiable or untriggerable constraints. It would be the case, for example, of a previously optional feature, that is also coincidentally the trigger of an untriggerable constraint, being transformed into mandatory in a new version of the feature model. In that case, the untriggerable constraint would become unsatisfiable.

Regardless of low prevalence, identification of these suspicious situations is very efficient, so there would be virtually nothing to be gained, performance-wise, by ignoring real-time detection of either untriggerable or unsatisfiable soft constraints in feature modeling editing tools.

Detection of contradictory pairs of soft constraints was likewise found to be efficient. This type of analysis needs to be conducted when a new soft constraint is added to the model, so that all contradictory pairs involving it are detected. Even the worst execution times found (just slightly over 0.5s), corresponding to contradiction detection in high CTCR models with a large number of soft constraints, are compatible with online operation in feature model editing tools (see Figure 7.4), so contradictory pair detection can be applied in feature model editing tools supporting soft constraints. This is fortunate, as results demonstrated that contradictory pairs of soft constraints are relatively common (see Figure 7.3). Clause density seems to be the predominant factor contributing to higher prevalence of contradictions. This is understandable, as a relatively high number of clauses (with respect to the number of features) may increase the chances that the introduction of a new soft constraint will intertwine the effects of two otherwise independent constraints.

### 7.4.2 Configuration Repair Testing

Although one single very and heavily cross-constrained model could not be successfully partitioned and handled by our algorithm, this was the single exception and we believe that it does not affect the validity and usefulness of our current approach, although this issue should be considered as a strong motivator

of future work. The test data used in our experiments includes the largest non-synthetic models available in the online repository. These include many examples that have been used independently as the unique or main case study for the validation of other works, such as the Smart House or Web Portal examples [MBC08, MRP<sup>+</sup>07], among others. The algorithm reveals very high performance, especially when considering iteration over alternative optimal repairs of a single configuration. Another case where algorithm performance stands out concerns repairs of multiple invalid configurations over the same feature model, as the most expensive operation needs to be performed only once per feature model. This is true even if an alternative selection criterion is used for each repair. This flexibility is unparalleled in other works we are aware of. Results for our experiments have consistently demonstrated the prevalence of multiple alternative optimal repairs, across the entire range of experiences, highlighting the usefulness of our problem-decomposition approach. The problem-decomposition approach has demonstrated the capability of compressing an exhaustive list of potential repairs (with the number of elements ranging from thousand to millions) into a manageable set of choices of modest dimension. Limitations of our approach include reduced capability of partitioning highly constrained feature models. In these instances, the consequences of poor partitioning include reduced scalability of the cover computation technique, and limited possibilities of decomposition of repairs into independent problems.

We found that, for most cases considered in this study, feature models can be effectively decomposed by our partitioning algorithm. A few highly constrained models are an exception to this. The extend to which this is problematic depends on other characteristics such as the model dimension.

The cover for the partitioned feature models can be computed efficiently, with the sole exception being the largest and more heavily constrained model. This exception can be explained by the poor partitioning achieved on that specific model. Nevertheless, other poorly partitioned models of more modest size were handled with reasonable efficiency, albeit comparatively lower with respect to other cases. In this way, it appears that the combined effect of large dimension and heavy constraint is necessary for significantly impairing cover computation efficiency.

Repairs for an invalid configuration can be computed efficiently. Results in Table C.5 strongly demonstrate very high performance when computing repairs: not only when identifying a first possible repair, but also when iterating over all other potential alternatives.

The problem-based decomposition is effective in reducing the number of repair options presented to the user. Considering it is strongly based on the partitioning of the feature model, it is bound to be less effective in those cases for which partitioning is less successful, such as the "DELL Computers" (see Table 7.8). Nevertheless, we found that a relevant reduction of the large repair lists could be achieved in all other models.

### 7.4.3 Empirical Testing of Enhanced Configuration Support

Our empirical experiments provided us with insights concerning our enhanced configuration support technique. The main goal of these tests was assessing efficiency and effectiveness gains. We expected to observe benefits in terms of the time required for creating the configuration, as well as improvements regarding soft constraint satisfaction rate. Additional qualitative feedback also helped better assessing our proposal. After conducting a statistical analysis of the collected data, benefits in terms of effectiveness were clearly apparent (more soft constraints were satisfied). However, similar benefits could not be observed in terms of efficiency. A possible explanation for this phenomenon, based on user feedback and observation of user behavior during the experience, is that at least some participants, regardless of the availability of configuration suggestions, did not refrain from exploring alternative configuration options, out of curiosity and the desire to explore the configuration space, even if ultimately the suggestions ended up being followed. This explanation also helps addressing an apparent inconsistency between qualitative and quantitative results: while no quantitative efficiency benefit was found, positive impact on efficiency (configuration time) was indicated in participant's feedback as being the factor that most gained from enhanced support. This is consistent with a scenario where the test participant rapidly identifies a good solution using the provided suggestions (thereby gaining the perception of enhanced efficiency), but then takes additional time exploring potential alternatives (nullifying the actual efficiency gains). Qualitative feedback was generically positive (see Figure 7.11), although some observations pointed out some difficulty interpreting the trade-off information. It was also pointed out that the configuration cases without enhanced support required better understanding of the feature model and constraints to achieve similar results.

Concerning the hypothesis in test, hypothesis 1 - *Enhanced configuration support reduces the time required to create the configuration*, could not be verified in the results. However, a statistically significant improvement in results was found,

confirming hypothesis 2 - *The constraint satisfaction performance improves when enhanced configuration is provided.*

## 7.5 Threats to Validity

In this section, we address several potential threats to the validity of our work.

- **Unrealistic Variability Model.** The first threat, common to different aspects of our work is restriction to variability modeling based on Boolean feature models. Other variability models such as Kconfig [ZC] and CDL [VD01] are preferably adopted in domains such as system software and have higher expressiveness. Nevertheless, Boolean feature models have a significant and relevant presence in SPL research and practice (via academic and commercial tool support). Extension to other variability models may be considered in future work.
- **Representativity of Test Cases.** Another related, but distinct threat to the validity of our work is concerned with the level of representativity of the test cases (feature models) considered in our experiments. It might be argued that the community-provided models of the S.P.L.O.T. repository do not have real world representativity and are improperly biased towards the low-end of the dimension, CTCR and/or clause density rate. Similar concerns have led to a certain trend where highly constrained models with very high feature count are artificially generated and chosen for testing purposes, rather than hand-crafted models. While this approach is certainly appropriate for assessing scalability, and while it is also true that in the operating systems domain model complexity may be significant, recent industrial surveys [BRN<sup>+</sup>13, BNR<sup>+</sup>14] have reported that feature models in actual use were found to be very lightly constrained or completely unconstrained, and thereby comparable to, or simpler than, those considered in our study.
- **Representativity of Soft Constraint Use.** Concerning our suspicious interactions detection, one potential threat is the representativeness of injected constraints. We address this by making sure that our soft constraint injection mechanism is based on plausible uses of soft constraints, and simpler cases of obviously degenerated constraints are explicitly avoided.
- **Scalability of Configuration Repair.** Another threat is scalability of our configuration repair approach. In this work, it is limited by cover optimization scalability and limited capability for partitioning the feature model.

This is partially mitigated by the fact that these factors impact mainly the cover-computation step of our approach, which needs to be run only once per feature model, but not subsequent repair generation. Also, this proved problematic with only the largest and most constrained model available in the repository. Nevertheless, this is an area for improvement. Ongoing and future work is focused on improving scalability by exploring the use of non-optimal covers, which does not compromise optimality of the generated repairs, by adopting a more aggressive partitioning strategy.

- **Reduced number of Participants in Empirical Study.** A threat to our empirical test is the relatively low number of participants. This concern is mitigated by the adoption of an within-subject approach, which has higher statistical power than an alternative between-group approach. In fact, we found that the experiment results provide statistically significant data demonstrating effectiveness improvements.





# Conclusions and Future Work

We begin the concluding chapter by revisiting the research questions in Section 8.1 and discuss them in the context of the content of earlier chapters. We proceed by presenting an overview of the evolution of this work (Section 8.2), from earlier inception efforts, and ensuing evolution leading up to the current thesis, and then beyond by proposing future research explorations derived from this line of work.

## 8.1 Research Questions Revisited

In Section 1.1 we presented the main research and detailed the research questions. The main research question,

**How to support the derivation of software products  
in SPL that best conform to stakeholders' goals?**

is decomposed in five sub-questions. The combined answers to these sub-questions globally address the main question, so we revisit them individually and discuss them in the next sections.

### **8.1.1 How to leverage the use soft constraints in SPL development to achieve this goal?**

Soft constraints can be introduced in the SPL lifecycle in both the domain engineering step or application engineering step. They are used to provide configuration suggestions and conflict detection and explanation. Further details are discussed in the following sub-sections.

#### **8.1.1.1 How can soft constraints be used in Domain Engineering?**

A configuration idealized by the stakeholder may be sub-optimal if other configurations exist that might better address his needs (see Section 1.1 and Figure 1.1). Sub-optimality may be caused by poor domain knowledge or underevaluation of the SPL possibilities. Soft constraints can be added to the variability model during domain engineering, that identify preferential or desirable properties of the configuration. This allows suggestions to be provided later during the configuration process (see Chapter 5) that help the stakeholder to overcome these difficulties. We pointed out in Section 4.1 that those preferential or desirable properties cannot be comfortably represented with standard hard constraints, which in effect leads either to improper modelling or even outright disregard of that information.

Having established the motivation for using soft constraints in domain modeling, we then identified three prototypical soft constraint application patterns in feature modeling (Section 4.2). We also point out in Section 4.2.5 that soft constraints can play a role in feature model evolution.

#### **8.1.1.2 How can suspicious soft constraint interactions be efficiently identified and reported to the Domain Engineer?**

In Section 4.3, we identified several types of suspicious soft constraint interactions. While these are not necessarily errors, it is doubtful that they are intentionally introduced by way of deliberated modeling action from the Domain Engineer. Therefore, it is useful to identify and report these interactions, so that the Domain Engineer can determine if any corrective action is required. We presented the identification procedures in Section 4.3, and conducted tests (Section 7.1) that demonstrated efficient performance compatible with real time operation in feature model editing tools.

#### 8.1.1.3 How common are these interactions?

Results in Section 7.1, obtained for publicly available feature models annotated with soft constraints injected according to usage patterns described Section 4.2, demonstrated variable degrees of prevalence. No unsatisfiable soft constraints were found, while untriggerable soft constraints were relatively uncommon. Contradictory constraints, on the other hand, were found to be relatively common. Although suspicious interactions are, globally, fairly uncommon, they can be very efficiently detected, and little performance gains are obtained by disregarding automated suspicious interaction detection.

#### 8.1.1.4 How to use soft constraints in the application engineering step?

Stakeholders' preferences do not necessarily conform to the feature model. For example, he may wish a product that includes features that are actually mutually exclusive. Also, while the stakeholder may have clear goals in mind, these may not necessarily translate into a single unique configuration. Therefore, he might have difficulty in pinpointing the exact configuration that best addresses those goals. An example would be the stakeholder desiring to include at least one among a certain set of optional features. Also, if the system is very complex, the internal consistency of the combined user preferences itself may be put into question. For example, the stakeholder may wish for web server support for at most one communication protocol for ease of management and security sake, while also requiring FTP support for file transfer and HTTP support for web hosting capabilities. By using soft constraints to represent stakeholders' preferences in the above scenarios, it becomes possible to represent the conflicting, inconsistent and incomplete specifications described in these scenarios.

Automated support, based on soft constraints, is thereby made possible. It provides configuration suggestions during application engineering. Conflicts are also identified, reported and explained to the user, providing him with improved understanding of the alternative trade-off possibilities and their consequences (see Chapter 5).

#### 8.1.2 How to provide enhanced configuration support?

The role of enhanced configuration support, in our work, is to facilitate the transformation of potentially invalid idealized configurations into realizable, valid configurations that are the best possible match to stakeholder's needs. A parallel concern is also helping the stakeholder to take the maximum possible advantage

of the possibilities of the SPL, by making sure that his idealized configuration is not sub-optimal, in the sense that another configuration exists that might better satisfy his needs.

One approach to offer enhanced configuration support is to leverage information from soft constraints. These can represent either domain properties or stakeholder's requirements. Available soft constraints are analyzed for two purposes.

The first is concerned with generating configuration suggestions, which steer the user towards configurations having desirable domain properties and that conform to his requirements. This helps addressing the issue of sub-optimality, as the suggestions provide to the user the opportunity to achieve desirable domain-related properties, that might not have been given due consideration previously.

On the other hand, the enhanced configurator is also capable of identifying conflicts and providing relevant feedback to the user. These conflicts are generated when simultaneous satisfaction of certain soft constraints is not possible, but alternative satisfaction is. In this case, it is up to the stakeholder to determine what is the preferable way of resolving the conflict. It bears mentioning that, while some constraints may be conflicting in all configurations, it is possible that a conflict manifests only in the presence of a certain partial configurations. Conflict resolution becomes then a context-dependent activity, in contrast with up-front anticipated resolution, in which the stakeholder may chose alternative resolutions for similar conflicts depending on specific product being created. For example, a soft constraint promoting parsimonious usage of memory resources may be more relevant in a mobile application than in a desktop application. Conflicts are identified and explained to the user via conflicted features, that is, open features whose configuration (either to a selected or deselected state) will be key for determining which conflicted soft constraints will be satisfied. An explanation is provided by indicating which constraints require selection of the conflicted feature, and which constraints require deselection. By using soft constraint satisfaction as the basis for conflict explanation, feedback is provided using high level concepts related to the requirements or domain properties, making it more accessible and understandable than alternatives such as unsatisfiable Boolean cores. Automatic identification and explanation of conflicts is key for assisting the user in achieving a viable configuration when the idealized configuration is invalid, while understanding the required trade-offs and respective consequences, in terms of feature selection.

Another strategy for aiding the stakeholder during the configuration process

is removing the barriers that impose step-wise specification of the configuration. The main problem with a step-wise specification approach, common in standard iterative configurators, is that the outcome becomes order dependent in overconstrained cases, due to choice propagation aiming to ensure a valid configuration is achieved. This can frustrate the user, as automated propagation of choices can result in implicit resolutions of trade-offs that may not be aligned with user's expectations. This may imply that backtracking becomes necessary, so that alternative configuration orders can be experimented with and alternative points of the configuration space explored. Although soft constraints mitigate this problem by providing additional orientation via suggestions and conflict explanation, the order dependency nevertheless persists. Prototype-based configuration, described in Chapter 6, generalizes iterative configuration to allow any number of features to be specified in a single iteration. Ultimately, the entire product may be specified in a single step. This obviates order-dependency issues impacting those features that are simultaneously specified. However, a subsequent resolution step becomes then necessary, because it may happen that the features are specified in a such a way it that does not correspond to any valid configuration. The resolution step analyses the proposed configuration, identifies defects and proposes possible solutions to the user. The resolution step is based on our cover-based configuration repair algorithm (Section 6.3). A key feature of this algorithm is a partitioning approach that allows a concise list of problems and possible repairs to be presented to the user, rather than an exhaustive list of every possible repair that might prove unwieldy, due to its dimension.

### 8.1.3 How to represent the user's idealized configuration?

Research question 3 asks "How to allow users to model, and inform the system, their idealized configuration, so that automated support can be given?". One important characteristic of idealized configurations is that they do not necessarily conform to the structure feature model or to the domain constraints. This precludes specification via the normal use of a standard iterative configurator, as these prevent, by design, invalid configurations from being inputted. We address this issue by using soft constraints to specify the idealized configuration of the stakeholder, fueling our enhanced configurator so that configuration suggestions and conflict analysis may be performed. However, another alternative is relying on the prototype-based configuration approach, which allows the stakeholder to introduce a complete (or partial) idealized configuration. This configuration is

subsequently analyzed so that, if needed, the required feedback, detailing possible alternative configuration repairs required to make it viable, is provided.

#### **8.1.3.1 How to handle possibly overconstrained and invalid idealizations?**

The mechanisms we propose offer support so that an overconstrained and invalid idealization can be realized, in such a way that the required changes have the smallest impact in terms of satisfaction of user's needs. By inputting a set of soft constraints describing the overconstrained or invalid configuration, overconstraintment will result in some those constraints failing to be satisfied. The conflict analysis mechanism will then identify conflicted features, that is, features whose selection status will determine which soft constraints will be alternatively satisfied. This allows the stakeholder to understand the consequences of selecting or deselecting that feature, so that he can understand the required trade off, in terms of soft constraint satisfaction.

The prototype-based approach can help the stakeholder in this regard by analyzing the provided configuration, and identifying the valid configurations that are as close as possible to the idealized. A set of repair actions is identified that, when applied, will transform the current configuration into one of the closest valid configurations.

While the soft-constraint based approach provides explicit identification of conflicts requiring trade-offs, who must each be addressed separately by the stakeholder, the prototype-based approach follows a more global approach by trying to identify the closest viable alternatives.

#### **8.1.3.2 How to address the large number of potential trade-offs?**

Research question 3b asks "How to properly address the large number of potential trade-offs that may be possible, without overwhelming the user with a very large number of alternative possibilities?".

Soft-constraint based support works in tandem with an iterative configurator. This means that, given features that are specified one at a time in each iteration, conflicts will manifest only gradually depending on the specific partial configuration that has been inputted so far. While many potential conflicts may exist, the stakeholder is confronted only with those that are actually relevant to the configuration being created.

If a prototype-based approach is allowed, then many features can be simultaneously specified in each single iteration. It is possible to take this approach to the ultimate consequences and specify the product in its entirety in a single

pass. In this case, the assumption is that the stakeholder is interested in realizing the configuration by finding a valid configuration that is "as close as possible" to the provided idealized configuration. Even with this assumption, which by itself prunes a lot of alternative possibilities, many different alternative repairs, with equal value, may be found (Section 7.2.3). To reduce the total number of alternatives presented to the user, we resort to a partitioning approach, that decomposes the repairs into actions required to address a certain number of problems (Section 6.4). Results in Section 7.2.4 demonstrate that significant improvements can be obtained in cases for which thousands or even millions of alternative repairs existed.

#### **8.1.4 How effective is enhanced configuration support?**

Effectiveness of enhanced configuration support is supported by both qualitative and quantitative results. In Section 7.3.5, user feedback shows that test participants considered that enhanced support helped them achieve higher constraint satisfaction ratios and better understand the required trade-offs. This is in accordance to the quantitative results of the experiment analyzed in Section 7.3.4, where a statistically significant improvement could be observed, in terms of constraint satisfaction.

#### **8.1.5 How efficient is enhanced configuration support?**

We ran several tests to assess the runtime efficiency of the proposed techniques (sections 7.1 and 7.2). Our suspicious interaction detection algorithms, described in Section 4.3, were found to be highly efficient. Our configuration repair algorithm, described in Chapter 6, also demonstrated its ability to efficiently partition the feature model and identify possible repairs. High performance is further highlighted by the fact that the most computationally expensive operation must be performed only once per feature model, rather than once per repair.

We also attempted to demonstrate gains of efficiency in terms of time required by the configurator to create the configuration according to a set of desired properties. Results were not conclusive. Although qualitative data indicated that efficiency gains were one of the most highly praised benefits perceived by test participants (Section 7.3.5), no statistically significant improvement could be found in our statistical analysis in Section 7.3.4. In our discussion of results, we provide a possible explanation for this discrepancy.

## 8.2 Past, Present, and Future

### 8.2.1 The Past

The foundations of this work began with the start of Ph.D. work in late 2007. The initial proposal was based on the study of feature/aspect interactions and some collateral derivations of that central theme [BM07, BM09a]. Although a grant was obtained that financed tuition fees in the initial year, no leave from service was obtained, so Ph.D. work and research was conducted in parallel with full time teaching (12 hours of lecturing/week) and also organizational responsibilities at Instituto Politécnico de Coimbra. Notwithstanding the institutional goodwill and support of my employers, this accumulation remained mostly a constant throughout the entire duration of this work, with exception of a single year (2009), when the original Ph.D. grant was replaced by a PROTEC grant, that offered a 50% reduction in lecturing schedule. Alas, the PROTEC program ended as abruptly as it started one year later.

Initial efforts in the first year tried to address the identification and resolution of order-dependency loops in the composition of software development assets. Subsequent ideas aimed at addressing composition order issues in aspect-oriented approaches. By interpreting an aspect as a statement of a property that was to be held in the final weaved target, it became possible to analyze whether or not a stable "steady state" composition was achieved, where the properties associated with each aspect were respected. Unfortunately, these ideas never really took off and developed beyond conceptualization efforts.

By 2009, attendance of tutorials concerning configuration knowledge management in SPL at the "Generative and Transformational Techniques in Software Engineering" (GTTSE) summer school, inspired the two earliest publications. In [BM09b], we proposed to address the complexity of configuration process by using a graphical method for describing configuration parameters dependencies and flow. This approach is centered on the use of configuration modules, which describe parameterizable implementation artefacts such as aspectual model components or source code files. This model describes how configuration information impacts the selection and configuration of features and their implementation artefacts. It provides the graphical tools akin to a template-based approach to configuration and implementation [CE00] by representing associations and specializations. However, it is not specific to any programming language or implementation technique. Additional capabilities are also provided such as the

representation of complex configuration dependencies via n-ary associations or support for specifying the enumeration of parameter lists. The capabilities of this representation technique can be observed in section A.1 of the appendices, where an example of one such model representing the configuration of a multimedia product is presented and described.

Another early work by the authors proposed an aspect-oriented modeling technique [BM09c]. It was designed to complement and support the work in [BM09b], by offering generic configurable aspects that can be instantiated, using a simple language, into concrete MATA aspects [WJE<sup>+</sup>09], capable of advising correctly diverse base models (see appendix A.2 for an overview of the proposed technique).

Although these were a promising start, further developments did not materialize and by the year 2010, we shifted our attention towards other topics that would later become a more central part of this work. We began considering the interesting potential of introducing soft constraints into the product line development cycle, and our initial exploration of that topic [BM11], in which the nature and semantics of Boolean soft constraints were discussed, was well received, garnering recognition in the form of a Best-Paper Award and the publication of an extended journal version in [BM12]. This further encouraged us to explore this line of research, and, as a natural consequence, the enhanced configuration support algorithms were first outlined in [BM13] and then later further extended and improved on [BM14b].

A different, but related line of research was however motivated, in the meantime, by simple exchanges with my advisor and research partners at UNL. A very simple question that often came up in discussion was the lack of provisions for simple freeform editing of existing configurations. Why wasn't it possible to simply change some features from selected to deselected, and vice-versa? It turns out, of course, that such a simple functionality is not so simple at all due to validity issues. However, it's certainly welcome from the perspective of the user, so it fit quite well with the ongoing line of research. We investigated existing approaches in the literature concerned with the reparation of invalid configurations, and quickly found that, while some previous work existed, some important properties were not addressed by existing solutions. So, we focused our efforts into developing a novel high performance configuration repair approach better suited for our specific needs [BM14a]. This algorithm also allowed overcoming

order dependency issues that are inherent to iterative configuration with step-wise specification of features, by enabling Prototype-based configuration to become a reality. Validation efforts with empirical experiments began in late 2014 and were completed by early 2015, with the remainder of the 2015 up to August being occupied with thesis writing. Simultaneously, other efforts are also underway, as described in Section 8.2.2.

### 8.2.2 The Present

Currently, some of the content of this work is undergoing preparation for journal publication. [BM15a] is an extended version of [BM14b], featuring an extended description of the algorithms found in Chapter 5 and including the results of the empirical assessment found in Chapter 7.

Other parts of this work are also amenable to additional publication. Content of Chapter 3, concerning the use of normative soft constraints, such as the generalized generalized impossibility function framework, generalizes and goes significantly beyond that which can be found in the original reference [BM11].

While the performance of the cover-based repair approach has been well established, additional empirical testing of the prototype-based configuration is also a possibility for the near future. This experiment would aim at evaluating perceived benefits, from the perspective of the user, offered by freeform specification and edition of a feature model configuration, powered by our cover-based repair algorithm. Publication of said results in a paper offering an extended discussion of prototype-based configuration and said results would naturally ensue [BM15c].

In summary, three new publications, with the following envisaged titles, are under preparation:

1. Journal paper. "Enhanced Configuration Support for Software Product Lines" [BM15a].
2. Conference paper. "Handling Exceptions to Constraints in Feature Models" [BM15b].
3. Journal/conference paper. "Prototype-based Product Derivation" [BM15c].

### 8.2.3 The Future

Once the short term projects, described in Section 8.2.2, are completed, new investigation possibilities, derived from or complementing this work, can be considered.

One possibility that suggests itself is broadening the range of applicable variability modeling techniques. While the current work is concerned with Boolean feature models, other modeling tools such as enhanced feature models, or alternative modeling approaches like CDL or Kconfig may be additionally considered. This entails additional challenges, including the handling of non-boolean constraints specified over numeric or textual data. Possible techniques that can be considered for application are constraint programming or multivalued-logic.

Another option for extending the scope of the work consists of leveraging historical configuration information as a potential source of additional configuration suggestions. This approach can work in tandem with other configuration techniques described here to further improve the configuration experience. To achieve this goal, techniques such as case-based reasoning might be employed. Recommendor systems are a related technology that also provide suggestions to their users using a mixture of domain and historical data sources, so exploration of similar approaches might be considered.

Other options for complementing this research include exploring other approaches for providing a user-centric product derivation experience. One such possibility includes advanced product preview techniques, which would address the problem of generating a product preview from the partial configurations generated during the configuration process. Having such previews available during all iterations of the configuration process would be useful, however constructing them for models with complex variability might be challenging, since only a fractional portion of the product features may be determined at a given time. The possibility of using available previews as a fundamental driver of the configuration system is an interesting possibility to consider. In this case, configuration could be achieved by progressively navigating through a tree of increasingly complete previews, dynamically generated at each point to indicate into what possibilities the current configuration might branch out.



# Bibliography

- [Bat05] Don S Batory. Feature Models, Grammars, and Propositional Formulas. In J Henk Obbink and Klaus Pohl, editors, *Proceedings of the 9th International Conference on Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20, Rennes, France, 2005. Springer.
- [Bat06] Don Batory. A tutorial on feature oriented programming and the ahead tool suite. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 3–35. Springer Berlin Heidelberg, 2006.
- [BB02] Michel Barbeau and Francis Bordeleau. A protocol stack development tool using generative programming. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin Heidelberg, 2002.
- [BBRC06] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Commun. ACM*, 49(12):45–47, December 2006.
- [BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.
- [BDNRG10] Ebrahim Bagheri, Tommaso Di Noia, Azzurra Ragone, and Dragan Gasevic. Configuring software product line feature models based on stakeholders’ soft and hard requirements. In *Proceedings of the*

- 14th International Conference on Software Product Lines: Going Beyond, SPLC'10*, pages 16–31, Berlin, Heidelberg, 2010. Springer-Verlag.
- [Beu01] Danilo Beuche. Feature Based Composition of an Embedded Operating System Family. In *Proceedings of the ECOOP 2001 Workshop on Feature Interaction in Composed Systems (FICS 2001)*, Budapest, Hungary, June 18-22, 2001, pages 55–60, January 2001.
- [BM07] Jorge Barreiros and Ana Moreira. Aspect interaction management with meta-aspects and advice cardinality. In *Aspects, Dependencies, and Interactions Workshop, as part of the 21st European Conference on Object Oriented Programming*, 2007.
- [BM09a] Jorge Barreiros and Ana Moreira. Managing features and aspect interactions in software product lines. In Kenneth Boness, João M. Fernandes, Jon G. Hall, Ricardo Jorge Machado, and Roy Oberhauser, editors, *SEDES Doctoral Symposium at the Fourth International Conference on Software Engineering Advances, ICSEA 2009, 20-25 September 2009, Porto, Portugal*, pages 506–511. IEEE Computer Society, 2009.
- [BM09b] Jorge Barreiros and Ana Moreira. A model-based representation of configuration knowledge. In *Proceedings of the First International Workshop on Feature-Oriented Software Development, FOSD '09*, pages 43–48, New York, NY, USA, 2009. ACM.
- [BM09c] Jorge Barreiros and Ana Moreira. Reusable Model Slices. In *Aspect Oriented Modeling Workshop @ ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems*, Denver, Colorado, 2009.
- [BM11] Jorge Barreiros and Ana Moreira. Soft Constraints in Feature Models. In *Proceedings of the Sixth International Conference on Software Engineering Advances*, pages 136–141. Xpert Publishing Services, 2011.
- [BM12] Jorge Barreiros and Ana Moreira. Soft Constraints in Feature Models: An Experimental Assessment. *International Journal On Advances in Software*, 5(3):252–262, 2012.
- [BM13] Jorge Barreiros and Ana Moreira. Configuration support for feature models with soft constraints. In Sung Y. Shin and José Carlos Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium*

- on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1307–1308. ACM, 2013.
- [BM14a] Jorge Barreiros and Ana Moreira. A cover-based approach for configuration repair. In Stefania Gnesi, Alessandro Fantechi, Patrick Heymans, Julia Rubin, Krzysztof Czarnecki, and Deepak Dhungana, editors, *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 157–166. ACM, 2014.
- [BM14b] Jorge Barreiros and Ana Moreira. Flexible modeling and product derivation in software product lines. In Marek Reformat, editor, *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, pages 67–70. Knowledge Systems Institute Graduate School, 2014.
- [BM15a] Jorge Barreiros and Ana Moreira. Enhanced configuration support for software product lines. (in preparation), 2015.
- [BM15b] Jorge Barreiros and Ana Moreira. Handling exceptions to constraints in feature models. (in preparation), 2015.
- [BM15c] Jorge Barreiros and Ana Moreira. Prototype-based product derivation. (in preparation), 2015.
- [BNR<sup>+</sup>14] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. Three cases of feature-based variability modeling in industry. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 302–319. Springer International Publishing, 2014.
- [BOBS89] Virginia E. Barker, Dennis E. O'Connor, Judith Bachant, and Elliot Soloway. Expert systems for configuration at digital: Xcon and beyond. *Communications of the ACM*, 32(3):298–318, March 1989.
- [BRN<sup>+</sup>13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A survey of variability modeling in industrial practice. In *Proceedings of the*

- Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13*, pages 7:1–7:8, New York, NY, USA, 2013. ACM.
- [Bry86] Randal Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, Aug 1986.
- [BSL<sup>+</sup>10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability modeling in the real. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 73, New York, New York, USA, 2010. ACM Press.
- [BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.
- [BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Using constraint programming to reason on feature models. In *Seventeenth International Conference on Software Engineering and Knowledge Engineering*, 2005.
- [BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002, September 1996.
- [CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative programming for embedded software: An industrial experience report. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002.
- [CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.
- [CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W Eisenecker. Staged Configuration Using Feature Models. In Robert L Nord, editor, *Proceedings of the 8th International Conference on Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004.

- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice*, page 2005, 2005.
- [Cle01] Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition, 2001.
- [CNM83] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1983.
- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.
- [CRM91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 181–186, New York, NY, USA, 1991. ACM.
- [CSW08] Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Sample Spaces and Feature Models: There and Back Again. In *Proceedings of the 12th International Conference on Software Product Lines*, pages 22–31. IEEE Computer Society, 2008.
- [CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Conference on Software Product Lines*, pages 23–34. IEEE Computer Society, 2007.
- [De 58] Augustus De Morgan. On the syllogism, no. iii, and on logic in general. *Transactions of the Cambridge Philosophical Society*, 10, 1858.
- [EFD05] Rudiger Ebendt, Görschwin Fey, and Rolf Drechsler. *Advanced BDD optimization*. Springer-Verlag US, 2005.
- [ELF08] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 99–108. IEEE, September 2008.

- [FFJ01] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering*, 15(2):165 – 176, 2001.
- [FJN<sup>+</sup>13] Alexander Felfernig, Michael Jeran, Gerald Ninaus, Florian Reinfrank, and Stefan Reiterer. Toward the next generation of recommender systems: Applications and research challenges. In George A. Tsihrintzis, Maria Virvou, and Lakhmi C. Jain, editors, *Multimedia Services in Intelligent Environments*, volume 24 of *Smart Innovation, Systems and Technologies*, pages 81–98. Springer International Publishing, 2013.
- [GFD98] Martin L. Griss, John Favaro, and Massimo D’Alessandro. Integrating feature modeling with the RSEB. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 76–85. IEEE Comput. Soc, 1998.
- [GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
- [GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
- [GSW89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in reiter’s theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, November 1989.
- [Ham50] Richard W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, XXIX(2), 1950.
- [Hou85] David Hounshell. *From the American System to Mass Production, 1800-1932: The Development of Manufacturing Technology in the United States*. ACLS Humanities E-Book. Johns Hopkins University Press, 1985.
- [Jan08] Mikolas Janota. Do SAT Solvers Make Good Configurators? In *Proceedings of the 12th International Conference on Software Product Lines*, pages 191–195, 2008.

- [JM11] Manu Jose and Rupak Majumdar. Cause clue clauses: Error Localization using Maximum Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, page 437, New York, New York, USA, 2011. ACM Press.
- [Jun04] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artificial Intelligence, AAAI'04*, pages 167–172. AAAI Press, 2004.
- [JZFF10] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems: An Introduction*. Cambridge University Press, 2010.
- [Kar53] Maurice Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.
- [KCH<sup>+</sup>90] Kyo C. Kang, Sholom Cohen, James Hess, William Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
- [KCH<sup>+</sup>92] Kyo C. Kang, Sholom G. Cohen, Robert R. Holibaugh, James M. Perry, and A. Spencer Peterson. A Reuse-Based Software Development Methodology (CMU/SEI-92-SR-004). Technical report, Software Engineering Institute, 1992.
- [KKL<sup>+</sup>98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, January 1998.
- [KLD02] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, July 2002.
- [LKL02] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools, ICSR-7*, pages 62–77, London, UK, UK, 2002. Springer-Verlag.

- [LMN08] Jaejoon Lee, Dirk Muthig, and Matthias Naab. An Approach for Developing Service Oriented Product Lines. In *2008 12th International Software Product Line Conference*, pages 275–284. IEEE, September 2008.
- [LS04] Inês Lynce and João Silva. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.
- [LSPS05] Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*. IEEE Computer Society, 2005.
- [MAK09] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. ACM Press, 2009.
- [MAW11] Fabiana G. Marinho, Rossana M.C. Andrade, and Cláudia Werner. A Verification Mechanism of Feature Models for Mobile and Context-Aware Software Product Lines. In *2011 Fifth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 1–10. IEEE, September 2011.
- [MBC08] Marcílio Mendonça, Thiago Bartolomei, and Donald Cowan. Decision-making coordination in collaborative product configuration. In *23rd Annual ACM Symposium on Applied Computing*, 2008.
- [MBC09] Marcílio Mendonça, Moises Branco, and Donald Cowan. S.P.L.O.T - Software Product Lines Online Tools. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*, page 761, New York, New York, USA, October 2009. ACM Press.
- [McC56] E.J. McCluskey. Minimization of boolean functions. *Bell System Technical Journal*, The, 35(6):1417–1444, Nov 1956.
- [Men09] Marcílio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models by*. PhD thesis, 2009.

- [Mil68] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.
- [MRP<sup>+</sup>07] João P. Morganho, Hugo Pimentão, Rita Ribeiro, Christoph Pohl, Andreas Rummler, and Ludger Schwanninger, Christa Fiege. Description of feasible industrial case studies. Technical report, AM-  
PLE Project - Aspect-Oriented, Model-Driven, Product Line Engineering, 2007.
- [NBD14] Mahdi Noorian, Ebrahim Bagheri, and Weichang Du. From intentions to decisions: Understanding stakeholders' objectives in software product line configuration. In Marek Reformat, editor, *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, pages 671–677. Knowledge Systems Institute Graduate School, 2014.
- [NE11] Alexander Nöhrer and Alexander Egyed. Optimizing user guidance during decision-making. In *14th International Conference (SPLC), Munich, Germany*, 2011.
- [NE13] Alexander Nöhrer and Alexander Egyed. C2o configurator: a tool for guided decision-making. *Journal of Automated Software Engineering (JASE)*, 20(2), 2013.
- [NEF03] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering*, number Xml, pages 455–464. IEEE Computer Society, 2003.
- [New90] Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, USA, 1990.
- [NNCI95] Victor P. Nelson, H. Troy Nagle, Bill D. Carroll, and David Irwin. *Digital Logic Circuit Analysis and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.
- [Par76] David L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

- [PBL05] Klaus Pohl, Günter Böckle, and Frank J. Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin Heidelberg, 2005.
- [Qui52] Willard V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):pp. 521–531, 1952.
- [Qui55] Willard V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):pp. 627–631, 1955.
- [RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *Integrated Design and Process Technology*, 2002.
- [Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987.
- [RF03] Silva Robak and Bogdan Franczyk. Modeling web services variability with feature diagrams. In Akmal B. Chaudhri, Mario Jeckle, Erhard Rahm, and Rainer Unland, editors, *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*, pages 120–128. Springer Berlin Heidelberg, 2003.
- [RP03] Silva Robak and Andrzej Pieczynski. Employing fuzzy logic in feature diagrams to model variability in software product-lines. In *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 305–311, 2003.
- [RRSK11] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.
- [Rud86] Richard L. Rudell. Multiple-Valued Logic Minimization for PLA Synthesis. Report UCB/ERL M86-65, University of California Berkeley, 1986.
- [SHTB06] Pierre Y. Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148. IEEE, September 2006.
- [SRI03] Detlef Streitferdt, Matthias Riebisch, and Technische Universität Ilmenau. Details of formalized relations in feature models using ocl.

- In *In Proceedings of 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003)*, pages 297–304, 2003.
- [SS07] Ida Solheim and Ketil Stølen. Technology Research Explained. Technical Report SINTEF A313, SINTEF, 2007.
- [STMS98] Timo Soininen, Juha Tiihonen, Tomi Männistö, and Reijo Sulonen. Towards a general ontology of configuration. *Artif. Intell. Eng. Des. Anal. Manuf.*, 12(4):357–372, September 1998.
- [Stu97] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2):111–125, April 1997.
- [TBD07] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering (ICSE’07)*, pages 44–53. IEEE, May 2007.
- [TBK09] Thomas Thüm, Don S. Batory, and Christian Kästner. Reasoning about edits to feature models. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 254–264. IEEE, 2009.
- [Tuk77] John Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.
- [VD01] Bart Veer and John Dallaway. The eCos Component Writer’s Guide. <http://ecos.sourceforge.org/ecos/docs-latest/cdl-guide/cdl-guide.html>, 2001.
- [vdS04] Tijs van der Storm. Variability and Component Composition. In *Software Reuse: Methods, Techniques, and Tools: 8th International Conference, ICSR 2004, Madrid, 2004*.
- [WBS<sup>+</sup>10] Jules White, David Benavides, Douglas Schmidt, Pablo Trinidad, Brian Dougherty, and Antonio Ruiz-Cortés. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094–1107, July 2010.
- [WJE<sup>+</sup>09] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira, and João Araújo. Mata: A unified approach for composing uml aspect models based on graph transformation. In Shmuel Katz, Harold Ossher, Robert France, and Jean-Marc Jézéquel, editors,

- Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *Lecture Notes in Computer Science*, pages 191–237. Springer Berlin Heidelberg, 2009.
- [WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [WPX<sup>+</sup>13] Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang. SmartFixer : Fixing Software Configurations based on Self-adaptive Priorities. In *Proceedings of the 17th International Software Product Line Conference*, pages 82–90, 2013.
- [WSO07] Hiroshi Wada, Junichi Suzuki, and Katsuya Oba. A Feature Modeling Support for Non-Functional Constraints in Service Oriented Architecture. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 187–195. IEEE, 2007.
- [XHSC12] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering*, pages 58–68. IEEE Press, 2012.
- [ZC] Roman Zippel and Contributors. kconfig language definition. <https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt>.

# Appendices





## Earlier Work

### A.1 Graphical Representation of Configuration Knowledge

In this section, we provide an illustrative example of application of the model-based representation of configuration knowledge described in [BM09b]. Consider the feature model represented in Figure A.1. The following configuration knowledge should be considered:

1. The "Send Photo" feature is dependent on the inclusion of the "Photo" feature (this information is not, for some reason, codified in the feature model).
2. If both "Photo" and "Video media options are selected then an additional menu must be included to allow for the user to switch between both media types. If only one media is present, no such functionality is required.

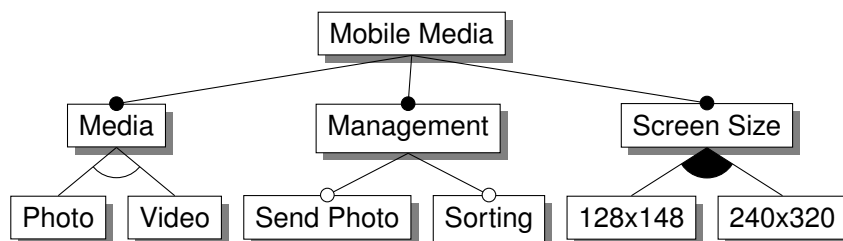


Figure A.1: Media application example

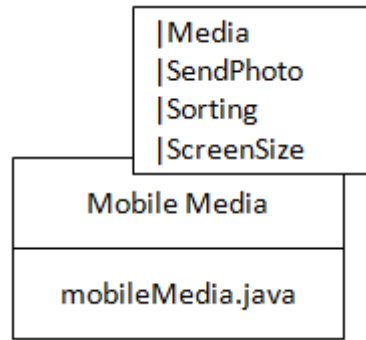


Figure A.2: Configuration module

3. For achieving acceptable performance, a 3rd party video decoder must be used in systems with higher resolution. In lower resolution systems, an in-house solution was found to be acceptable and is to be used instead, to reduce royalty costs.

Configuration of the feature model entails selecting one of the alternative screen sizes, selecting which media options will be available, and whether the send photo or sorting services are required. This can be represented in our model with the following *configuration module* (CM) shown in Figure A.2: Each CM may correspond to a feature of the model, but that is not always the case as shown bellow. A CM may include a set of parameters. In this case, the "Mobile Media" CM parameters are "Media", "Send Photo", "Sorting", and "Screensize". These parameters are specified by the user to describe the desired configuration. For example, (Media="Photo,Video", SendPhoto="true", Sorting="false", and ScreenSize="128x148"). As shown, parameters may contain single or multiple values. Specifying the parameters for this CM will instantiate it and trigger the inclusion of the mobileMedia.java source code file into the project.

Our model allows describing the propagation of configuration information through the CMs. These associations describe both the flow of configuration data and cardinality constraints that must be respected when instantiating the CMs. Figure A.3 describes how configuration from the "Mobile Media" CM is to be propagated to the Media CM. The arrow indicates how the configuration information flows, and "\*" represents iteration. "| MediaType= | Media\*" indicates that the MediaType parameter (of "Media" CM) will successively assume each one of the values of "Media" parameter of the "Mobile Media" CM (rather than being assigned the same list of values which would be achieved by the straightforward "| MediaType= | Media"). This will cause the Media CM to be instantiated twice: once for "MediaType=Photo" and yet again for "MediaType=Video".

Composition can be used as a shorthand notation for indicating that similarly

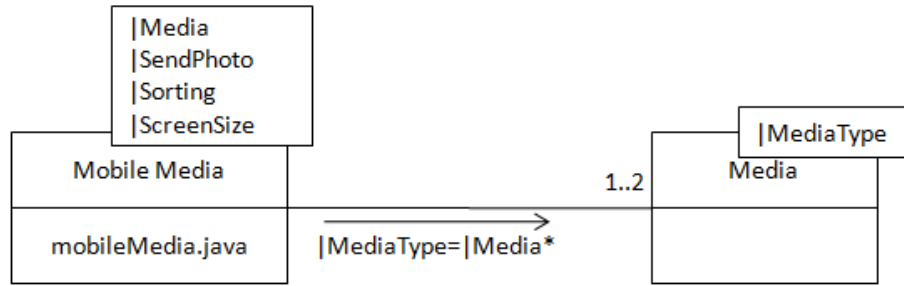


Figure A.3: Association describing configuration flow and cardinality constraints

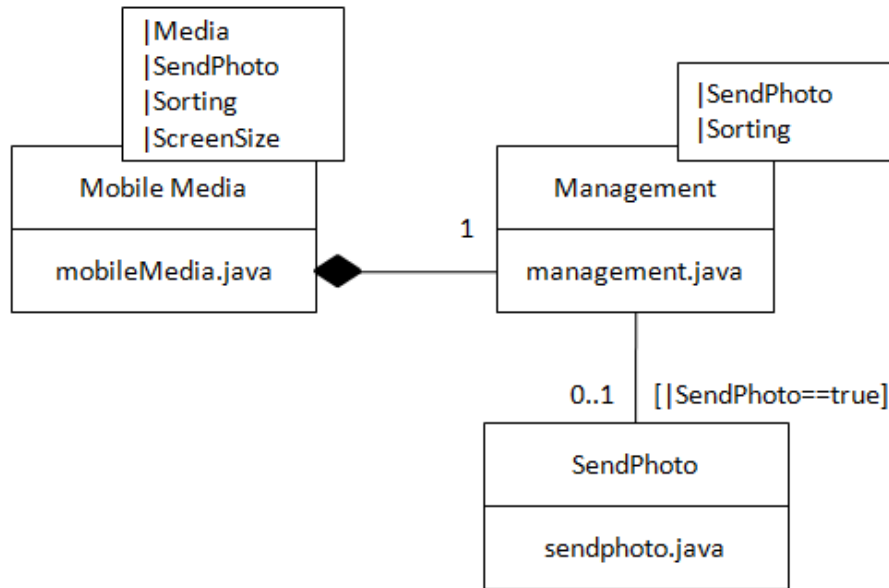


Figure A.4: Composition example

named parameters on either side of the association should be set to the same value. A guard can also be used to indicate under what circumstances an associated CM should also be instantiated. This is illustrated in Figure A.4.

A specialization of a configuration module is a specific implementation of that configuration module that is to be used whenever a specific value or set of values is used in the parameters (Figure A.5).

Figure A.6 takes advantage of the specialized configuration of "Media" to represent the constraint requiring "Photo" if "Send" service is selected.

Figure A.7 illustrates the use of n-ary associations to describe the inclusion of an anonymous CM that ensures that the source code file containing the required menu is built into the application.

Finally, Figure A.8 wraps up our example by using specializations to describe the variable implementation depending on selected resolution.

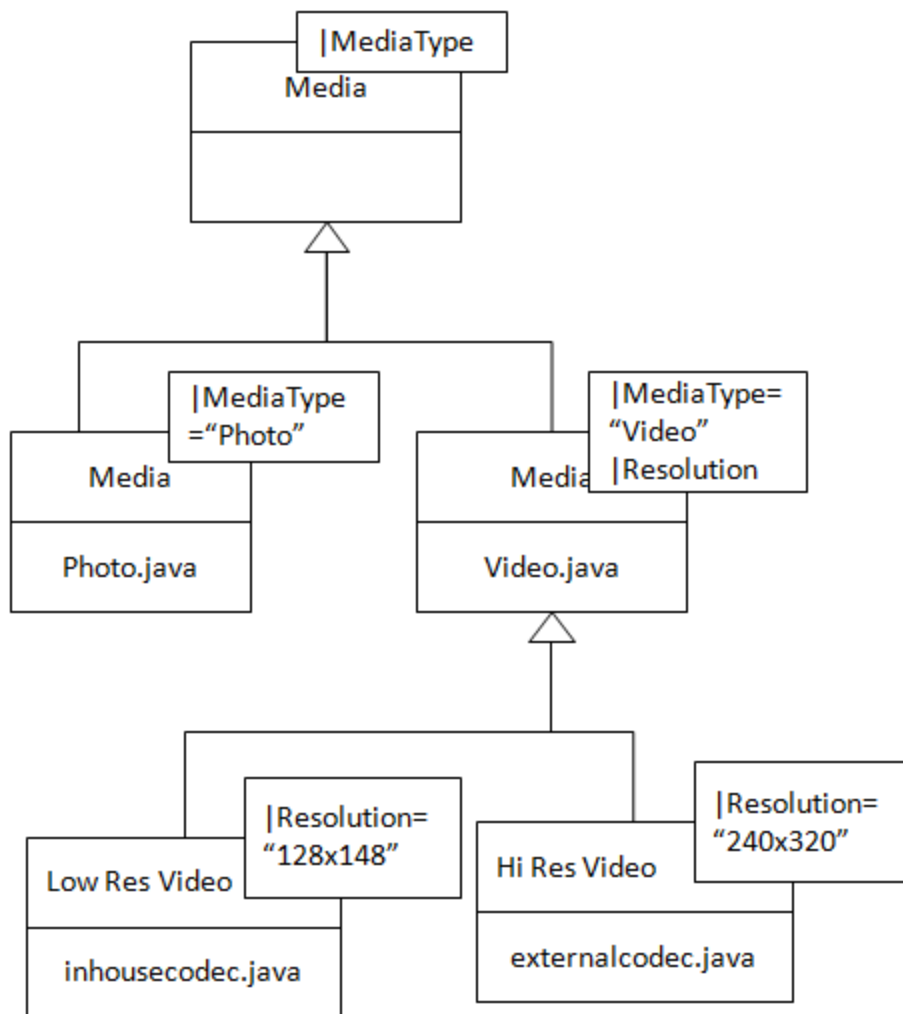


Figure A.5: Specialization example

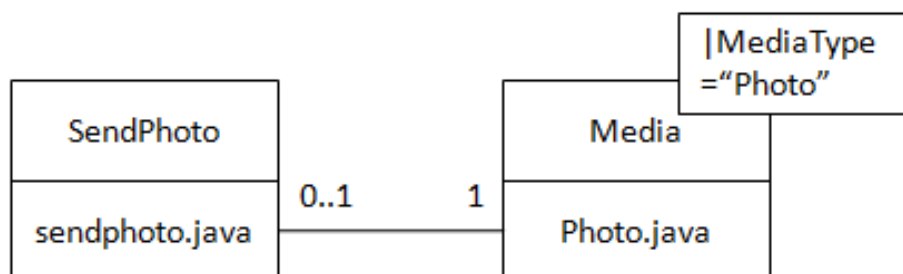


Figure A.6: Association describing constraint only, without any configuration flow.

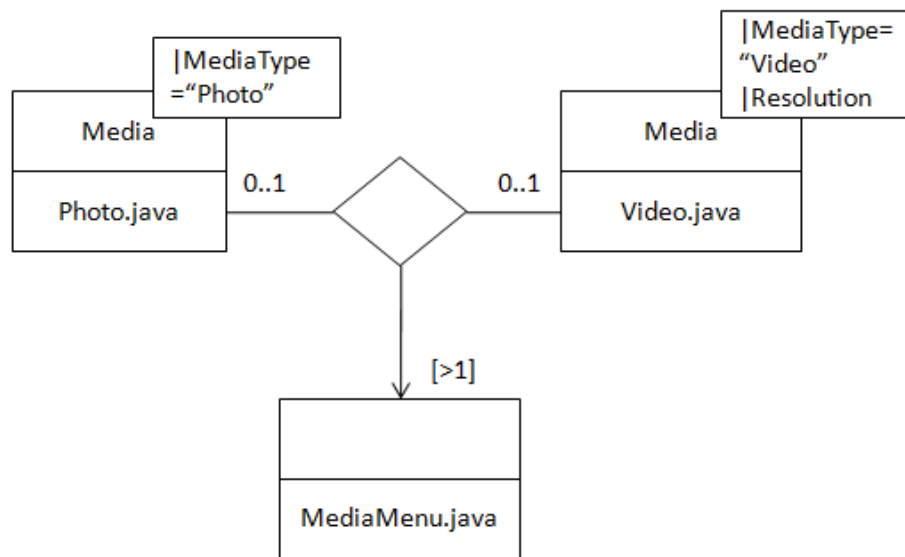


Figure A.7: n-Ary association to represent complex constraint

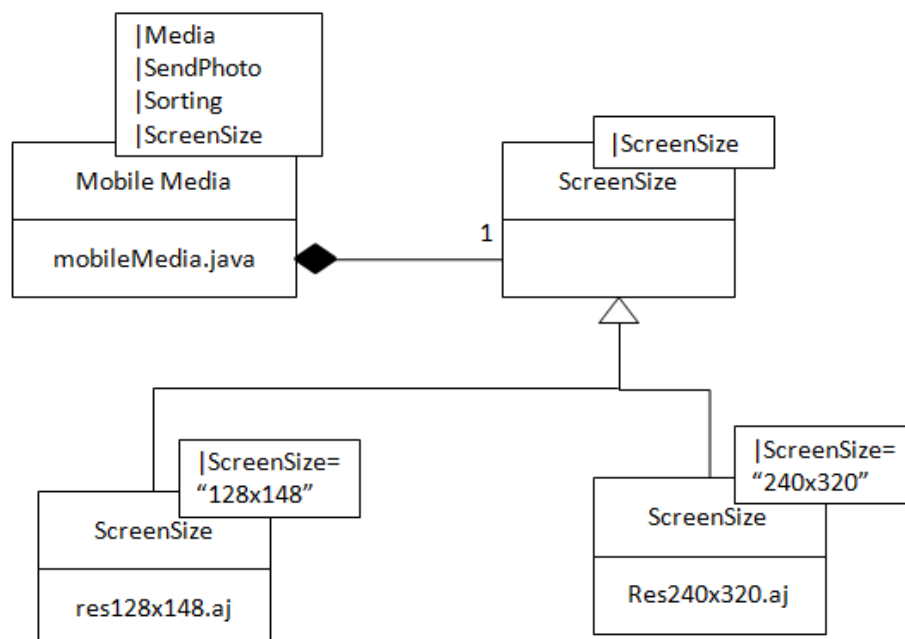


Figure A.8: Representing alternative implementations with specializations.

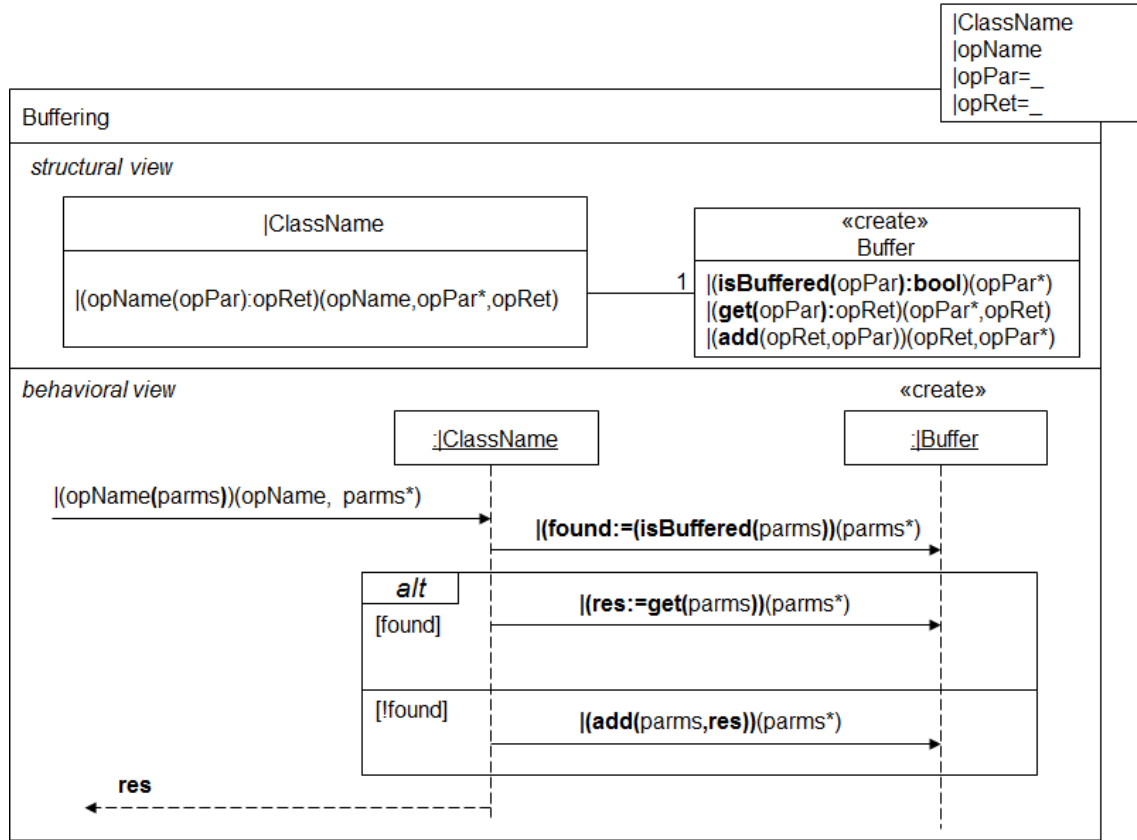


Figure A.9: Example of aspectual model slice

## A.2 Reusable Model Slices

### A.2.1 Overview

This section provides a brief illustration of the Resusable Model Slices (RMS) aspect-oriented modelling technique described in [BM09c]. The main objective of this approach is providing *configurable* aspectual models (the model slices), that can be tailored to apply to different base models. It is well suited to implement the model-based configuration representation described in Section A.1. Figure A.9 provides an example of an aspectual model slice, whose contents are fully explained over the next subsections. This model slice describes the buffering of an operation's results: it is named "Buffering", has a structural and behavioral description in two separated containers (using class and sequence diagram slices, respectively). This model slice is generalized by parameters corresponding to generic variables `ClassName`, `opName`, `opPar` and `opRet`. To create a concrete model slice, an instance of a RMS should be constructed by supplying the required parameters. An instance of RMS is a MATA aspectual model pattern

[WJE<sup>+</sup>09] specifically tailored for making the changes specified via the configuration. Configuration is accomplished by a very simple language. An instance is defined as:

```
RMSidentifier(Parameter1, Parameter2, ...)
```

For example, to create concrete model slices that add buffering to operation `getPerson(name:String, age:Integer):Person` from class `Factory`, an instance of the Buffering RMS could be created like this (parameter are specified in the same order as declared in the RMS):

```
Buffering (Factory, getPerson, "name:String,  
age:Integer", Person)
```

### A.2.2 Wildcards

The special symbol `"|_"` may be used to represent "any matching element". When used within the generalized transformations, the same symbol should be interpreted as "don't care". For example, to create concrete model slices that add buffering to any operation `getPerson` that returns a `Person`, regardless of the containing class or operation arguments, one could create the following instance:

```
Buffering(|_, getPerson, |_, Person)
```

### A.2.3 Default values

Parameters may be given default values. In Figure A.9, parameters `opPar` and `opRet` have the `"|_"` special symbol (any matching element) as the default value. This means that any such parameter will be assigned that value unless otherwise specified. For example, in Figure A.9, both the `opPar` and `opRet` parameters have the `"|_"` default value. Taking this into consideration, concrete model slices that add buffering to any operation `f` in class `Test`, regardless of parameters or return type might be obtained by instantiating the "Buffering" RMS as:

```
Buffering (Test, f)
```

### A.2.4 Operations

A special syntax was created for conveniently manipulating and generalizing operations within the generalized transformations. An operation is represented as

```
| (StringLiteral) (NonTerminalTokenList)
```

where `StringLiteral` is a description of the desired operation, using concrete syntax as desired. Within this description, string tokens may be used to describe non-terminal generic symbols that are to be considered variable model elements. These are explicitly identified in the non-terminal token list, to distinguish them from the literal symbols. As an example:

```
| (getPerson ( personKey ) :Person) (personKey)
```

This pattern will match any operation with name `getPerson` that returns a `Person` and has a single parameter of any type. The `personKey` token is identified as being a non-terminal token, being bound to whatever the actual parameter is. For example, this pattern will match both:

```
getPerson(Integer) :Person
getPerson(String) :Person
```

with `personKey` being bound to `Integer` and `String`, respectively. If an `*` is appended to a non-terminal token identifier, then it will match any list of operation parameters, including a void list. The next example will match any operation with the corresponding name and return type, regardless of the number or type of parameters:

```
| (getPerson ( personKey ) :Person) (personKey*)
```

If the actual parameter list is not referred to elsewhere in the RMS, then the previous example could be rewritten as:

```
| (getPerson (|_* ) :Person) ()
```

### A.2.5 Instantiation

A RMS is instantiated by replacing all provided template parameters in the generic MATA models as appropriate. If some generic variables remain undefined in the model (or are set to "don't care") then these will be matched against the base model. All matching parameter bindings are identified and each such binding will be used to create an alternate transformation. As such, in a single RMS instance, each generic transformation may originate multiple transformations. As an example, the instance:

```
Buffering(|_, getPerson, |_, Person)
```

will generate multiple parameter bindings to match any class that has a `getPerson` operation with `Person` return type. If this operation is overloaded in some classes, then additional parameter bindings will be used to reflect that as well. The MATA transformations thus generated will be then applied to the base model. Dependencies and conflicts between these individual transformations will be handled as usual by the MATA framework. As an example, the instance:

```
Buffering( Sale, computeTax, "", Money)
```

would result in the concrete MATA transformations illustrated in Figure [A.10](#).

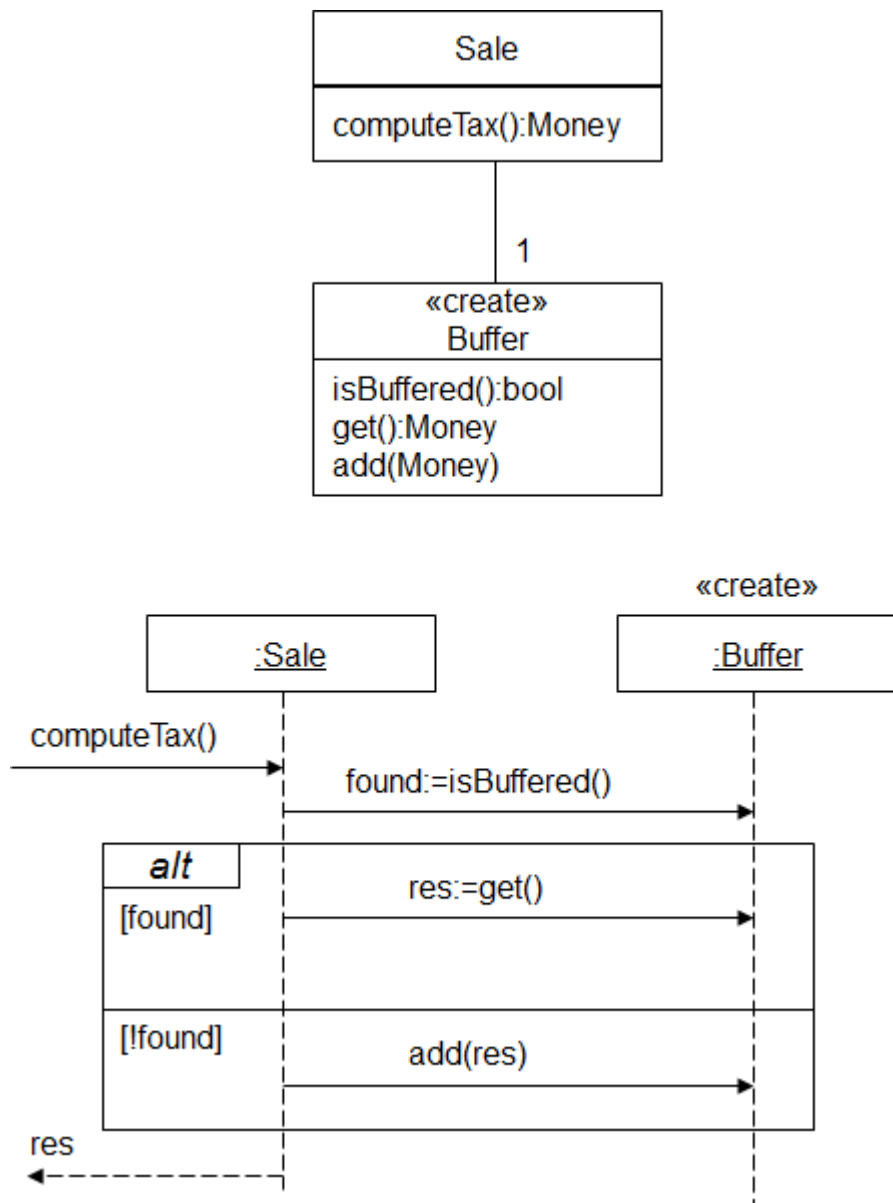


Figure A.10: Resulting MATA transformations



# Box and Whiskers Plots

## B.1 Box and Whiskers Plots

Most of our results are presented using box and whiskers plots, devised by Tukey [Tuk77] in the 1970's. They offer a compact summary of the distribution of a data set. In these plots, a box is always used to represent the first, second and third quartiles of the data ( $Q_1, Q_2, Q_3$ ).  $Q_2$  corresponds to the median. The dimension of the box ( $Q_3 - Q_1$ ) represents the interquartile range ( $IQR$ ), a statistical measure of the dispersion of the data. It is a robust measure, unlike others such as standard deviation or variance, who can be greatly affected by outliers. Whiskers are line segments placed above and below the box, whose extremities (known as the *fences*) can be associated with different statistics. We follow Tukey's approach of using the high and low fences to denote the highest datum below  $Q_3 + \frac{3}{2}IQR$  and the lowest datum above  $Q_1 - \frac{3}{2}IQR$ , respectively. Values outside these ranges are considered outliers. Outliers are represented explicitly using dot markers above/below the whiskers. White or open dots are used for normal outliers, while black dots are used for extreme outliers (values above  $Q_3 + 3IQR$  or below  $Q_1 - 3IQR$ ). We also identify the mean of the data with an 'x' marker. Figure B.1 shows an example of a box and whiskers plot. Two data series are represented side by side. The  $IQR$  for both series is represented by the height of the boxes: 45 for the left-side series and 20.5 for the other. This information can be perceived from the boxplot at a glance. The top and bottom of the boxes correspond to

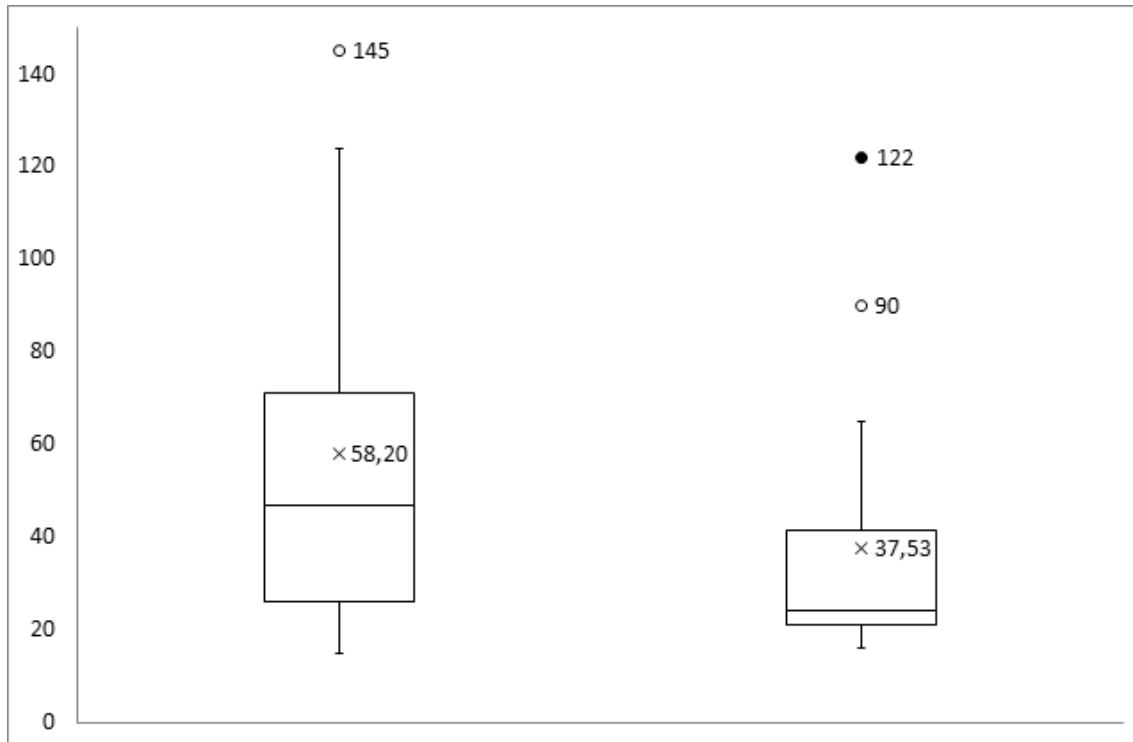


Figure B.1: Box and whiskers plot example

the third and first quartile ( $Q_3$  and  $Q_1$ ), respectively, whilst the box divider indicates the median values ( $Q_2$ ), 47 and 24. The series on the left side of the plot has average 58.20, while the other series as an average of 37.53. These values are marked with 'x'. Although the mark is labeled with the actual average value, we generally avoid doing so to avoid visual pollution of the chart.

The whiskers extend up from the top of the boxes, up to the highest datum still within  $1.5IQR$  distance of  $Q_3$ . Conversely, they also extend down to the lowest datum still within  $1.5IQR$  of  $Q_1$ . This offers a clear picture of the symmetry of the distribution (or lack thereof) around the median. In this case, it is apparent that both series are skewed towards the highest values. The whisker limits also represent the range beyond which datum are classified as outliers. These are shown plotted above both whisker lines. The values marked with a clear circle are normal outliers, while the value marked with a black dot is an extreme outlier (because it is further than  $3IQR$  away from  $Q_3$ ). Outliers are sometimes labelled with an identification and explanation of that particular result.

Outliers represent "interesting" data in that it falls outside the expected range. Outliers can occur by chance alone, especially if the population is very large, but they are often indicative of either measurement problems or low kurtosis ("peakedness"). The first case suggests removal of the offending points from the

data set, while the second suggests that additional insight should be sought.





## **Validation Results**

Table C.1: Results for soft constraint injection

<b>Model Name</b>	<b>Nr. of Constraints Injected</b>
AndroidSPL	21
Arcade Game PL	48
bCMS system	22
Billing	105
Car Selection	50
Consolas de Videojuegos	18
OS	14
DATABASE_TOOLS	28
DELL Computers	119
Documentation_Generation	20
DS Sample	6
Electronic Drum	12
E-science application	25
HIS	20
Hotel Product Line	38
J2EE web architecture	38
Letovanje	18
Linea de Experimentos	28
Meshing Tool Generator	36
Model_Transformation	37
FraSCAti	87
MFP	81
Meeting Config	22
Printers	29
Eclipse1 - Reuso	45
Smart Home	40
Smart Home v2.2	38
SmartHome_vConejero	36
Jogo	22
Thread	19
Video Player (a)	27
Video Player (b)	17
Web_Portal	29
xtext	94
Coche ecologico	94
Webmail	81
BankingSoftware	176
Estrutura_Decisiones	366
BattleofTanks	144

Table C.2: Results for identification of untriggerable and unsatisfiable soft constraints

<b>Model</b>	<b>Unsat. Constraints</b>	<b>Untrigger. Constraints</b>	<b>Total Time (ms)</b>	<b>Average Time (per constr.) (ms)</b>
AndroidSPL	0	0	28	1,33
Arcade Game PL	0	0	373	7,94
bCMS system	0	0	195	13,00
Billing	0	9	485	4,62
Car	0	0	303	6,06
Consolas de Videojuegos	0	0	22	1,22
OS	0	0	15	1,07
DATABASE_TOOLS	0	0	39	1,50
DELL Computers	0	0	605	5,08
Documentation_Generation	0	0	25	1,25
DS Sample	0	0	15	2,50
Electronic Drum	0	0	24	2,00
E-science application	0	1	47	1,88
HIS	0	1	38	1,90
Hotel Product Line	0	0	55	1,45
J2EE web architecture	0	0	61	1,65
Letovanje	0	0	25	1,39
Linea de Experimentos	0	0	27	1,50
Face Animator	0	2	59	1,64
Model_Transformation	0	0	81	2,25
FraSCAti	0	0	230	2,64
MFP	0	2	199	2,46
Meeting Config	0	0	33	1,50
Xerox	0	0	212	7,31
Eclipse1 - Reuso	0	1	59	1,31
Smart Home (a)	0	0	46	1,21
Smart Home (b)	0	0	38	1,00
SmartHome_vConejero	0	2	58	1,61
Jogo	0	0	36	1,64
Thread	0	0	9	1,29
Video Player (a)	0	0	31	1,15
Video Player (b)	0	0	25	1,47
Web_Portal	0	0	34	1,17
xtext	0	1	288	3,06
Coche_ecologico	0	0	362	12,07
Webmail	0	0	138	3,94
BankingSoftware	0	1	573	6,10
Estrutura_Decisiones	0	0	302	13,73
BattleofTanks	0	0	228	10,36

Table C.3: Results for identification of contradictory pairs of soft constraints

Feature Model	Soft Constraints	SC Pairs	Contradictions	Time [ms]
AndroidSPL	21	210	17	167
Arcade Game PL	48	1128	9	3451
bCMS system	22	231	27	251
Billing	105	5460	913	15311
Car Selection	50	1225	79	2158
Consolas de Videojuegos	18	153	3	104
OS	14	91	12	39
DATABASE_TOOLS	28	378	19	266
DELL Computers	119	7021	5727	40034
Documentation_Generation	20	190	9	107
DS Sample	6	15	3	20
Electronic Drum	12	66	3	52
E-science application	25	300	17	357
HIS	20	190	6	176
Hotel Product Line	38	703	29	383
J2EE web architecture	38	703	31	761
Letovanje	18	153	27	111
Linea de Experimentos	28	378	12	230
Face Animator	36	630	165	434
Model_Transformation	37	666	11	1012
FraSCAti	87	3741	177	7894
MFP	81	3240	1605	9594
Meeting Config	22	231	18	178
Xerox	29	406	2	1093
Eclipse1 - Reuso	45	990	114	1005
Smart Home (a)	40	780	40	478
Smart Home (b)	38	703	33	456
SmartHome_vConejero	36	630	130	397
Jogo	22	231	8	243
Thread	19	171	30	99
Video Player (a)	27	351	15	262
Video Player (b)	17	136	3	101
Web_Portal	29	406	43	204
xtext	94	4371	291	14732

Table C.4: Partitioning and cover extraction results

Model Name	Partitions	Number of Terms	Partitioned Terms	Time (ms)
Arcade_Game	16	5418	5433	3652
Dell_Computers	1	853	853	471
Coche_ecologico	44	207360	75	1357
Meeting_Config	26	138240	51	673
xtext	25	5231304	127	664
Eclipse1-Reuso	7	990000	99	220
Car	16	10777536	72	559
Smart_Home (a)	18	583680	60	530
MFP	12	5	16	418
Smart_Home (b)	20	23232	149	632
ConsolasVideojuegos	4	79200	112	722
OS	10	8400	31	416
Video_Player (a)	19	50400	43	516
FaceAnimator	10	8640	44	259
AndroidSPL	14	2304	57	324
FraSCAti	14	1760	455	396
Xerox	49	8707129344000	144	1404
bCMS_system	37	3072	54	893
J2EE web architecture	29	5225472	57	901
DS Sample	6	6912	34	220
Letovanje	4	12456	186	133
Hotel_Product_Line	34	2592	47	1017
database_tools	14	2177280	67	372
Electronic_Drum	16	331776	41	410
Jogo	23	184320	43	607
Video_Player (b)	28	93632	72	786
Model_Transformation	22	207152640	165	599
his	39	560	53	927
Documentation_Generation	5	15162	2534	586
thread	1	390	390	83
Web_Portal	5	14220	300	129
BankingSoftware	66	32060448	321	1747
BattleofTanks	14	42996610800	130	392
Billing	19	24	42	579
Estructura Decisiones	3	-	-	-

Table C.5: Repair results - Finding repairs that minimize Hamming distance from random invalid configurations.

Model Name	Average Number of Repairs	Time First[ms]	Iteration over next (max. 1k)[ms]
Arcade_Game	1.70	28.54	0.33
Dell_Computers	19.63	5.19	0.85
Coche_ecologico	75.53	0.18	1.22
Meeting_Config	35.93	0.12	0.22
xtext	26.77	0.53	0.14
Eclipse1-Reuso	30.47	0.19	0.32
Car	912.80	0.14	0.10
Smart_Home (a)	6.77	0.12	0.25
MFP	1.60	0.09	0.34
Smart_Home (b)	5.27	0.30	3.49
ConsolasVideojuegos	25.03	0.19	0.35
OS	17.83	0.06	0.31
Video_Player (a)	2.80	0.09	1.99
FaceAnimator	6.10	0.10	12.26
AndroidSPL	8.00	0.11	0.14
FraSCAti	4.73	3.61	0.08
Xerox	7166361626.90	0.36	1570.29
bCMS_system	9.00	0.18	0.04
J2EE web architecture	55.40	0.11	3.75
DS Sample	305.07	0.10	4.26
Letovanje	22.93	0.48	0.05
Hotel_Product_Line	11.57	0.08	0.40
database_tools	6.47	0.15	0.31
Electronic_Drum	2918.00	0.07	0.15
Jogo	1062.27	0.09	4.80
Video_Player (b)	22.20	0.16	0.06
Model_Transformation	104.07	0.31	1.41
his	8.23	0.13	0.33
Documentation_Generation	10.53	8.68	3.42
thread	7.33	1.96	0.88
Web_Portal	5.17	0.51	0.55
BankingSoftware	43.87	3.23	0.16
BattleofTanks	371819.60	0.46	9.51
Billing	2.33	0.56	8.50

Table C.6: Repair results - Finding repairs for invalid configurations preserving selected features.

<b>Model Name</b>	<b>Average Number of Repairs</b>	<b>Time First [ms]</b>	<b>Iteration over next (max. 1k) [ms]</b>
Arcade_Game	1.10	9.33	0.07
Dell_Computers	6.73	1.72	0.14
Coche_ecologico	28.50	0.33	0.09
Meeting_Config	6.47	0.19	0.02
xtext	1.00	0.40	0.02
Eclipse1-Reuso	5.00	0.23	0.03
Car	40545.60	0.31	0.03
Smart_Home (a)	1.27	0.17	0.05
MFP	1.03	0.13	0.08
Smart_Home (b)	1.43	0.23	0.09
ConsolasVideojuegos	13.43	0.14	0.07
OS	3.20	0.08	0.07
Video_Player (a)	1.67	0.11	0.30
FaceAnimator	2.83	0.11	0.73
AndroidSPL	105.93	0.72	0.04
FraSCAti	1.90	0.70	0.30
Xerox	20.77	0.33	60.58
bCMS_system	3.30	0.16	0.02
Experimento	1.00	0.14	0.95
J2EE web architecture	3.33	0.15	0.05
DS Sample	76.53	0.13	33.77
Letovanje	10.23	0.27	0.02
Hotel_Product_Line	4.30	0.14	0.09
database_tools	2.97	0.16	0.02
Electronic_Drum	131.73	0.11	1.36
Jogo	20.60	0.12	1.76
Video_Player (b)	1.00	0.15	0.02
Model_Transformation	34.90	0.24	0.58
his	28.93	0.19	0.08
Documentation_Generation	3.33	2.87	1.34
thread	3.63	0.73	0.02
Web_Portal	29.20	0.33	1.05
BankingSoftware	65.33	0.81	0.02
BattleofTanks	647985.20	0.27	5.49
Billing	1.50	0.23	0.05





# **GQM Document**

Goal		Purpose Issue Object (process) Viewpoint	Evaluate the effectiveness and efficiency and of enhanced configuration support from the perspective of the application engineer / developer
Question (Q1)			What is the efficiency of the standard configuration process?
Metrics		M1	Configuration time (without support)
Question (Q2)			What is the efficiency of the enhanced configuration process?
Metrics		M2	Configuration time (with support)
Question (Q3)			What is the effectiveness of the standard configuration process?
Metrics		M3	Constraint satisfaction rate (without support)
Question (Q4)			What is the effectiveness of the enhanced configuration process?
Metrics		M4	Constraint satisfaction rate (with support)
Question (Q5)			Does the efficiency of the configuration process improve?
Metrics		M5 M6	M2/M1 Subjective evaluation of developer
Question (Q6)			Does the effectiveness of the configuration process improve?
Metrics		M7 M8	M12/M11 Subjective evaluation of developer
Question (Q7)			Does configuration support help understanding the required trade offs?
Metrics		M9	Subjective evaluation of developer

M1 – Configuration Time is measured from the time the user begins a configuration task without support, up until the moment where it ends. The configuration task begins when the

user opens the corresponding feature model, and ends when the user marks the configuration as done (advancing to the next configuration).

M2 – Similar to M1, for configuration tasks with soft constraint support.

M3 – The ratio of satisfied soft constraints without support.

M4 – The ratio of satisfied soft constraints with support.

M5 – The ratio of the configuration times, with and without support.

M6 – The test subject is asked if he feels configuration support helped him achieve complete the configuration faster. The answer is provided in a 5 point discrete scale.

M7 – The ratio of the configuration efficiency, with and without support.

M8 – The test subject is asked if he feels configuration support helped him achieve complete the configuration more efficiently. The answer is provided in a 5 point discrete scale.

M9 – The test subject is asked if he feels configuration support helped him better understand potential trade-offs. The answer is provided in a 5 point discrete scale.





## **Experiment Test Cases**

## Scenario Name: Web Portal

### Description:

This feature model describes possible configurations of a web server, allowing specification of supported protocols, security features and additional services .

**You are requested to create a configuration following these recommendations, if possible (it may be impossible to simultaneously satisfy all these constraints):**

- *We wish to support the lowest possible number of protocols.*
- *Performance should be in the range of milliseconds or seconds (Ms or Sec features)*
- *Logging of server operations would be desirable*
- *Secure data transfer (Data\_Transfer feature) is also considered an useful feature to include.*

## Scenario Name: fraSCAti

### Description:

The OW2 FraSCAti Software Product Line (SPL) allows users to build highly "à la carte", configurable and extensible Service Component Architecture (SCA) runtime platforms according to both their application requirements and target system constraints. This feature model is a compact representation of all OW2 FraSCAti features and their constraints, which captures all possible OW2 FraSCAti configurations. Each OW2 FraSCAti feature is a distinctive user-visible plugin of the OW2 FraSCAti SPL (e.g. Web Service binding support, BPEL implementation support, a Java compiler used).

**You are requested to create a configuration following these recommendations, if possible (it may be impossible to simultaneously satisfy all these constraints):**

***To keep things as simple as possible:***

- *Support the lowest possible number of metamodels.*
- *No additional implementations other than the mandatory Composite and Java (Implementation\_Composite and Implementation\_Java) should be included. The fewer the better.*
- *No additional Bindings other than the mandatory SCA binding (Binding\_SCA feature) should be included.*
- *Each and every single one of these features is desired: JMX, BindingFactory, FScript, RemoteManagement, Explorer.*

### **Scenario Name: DELL LAPTOP/NOTEBOOK**

#### **Description:**

This feature model describes a manufacturer's line of computers. It represents possible combinations of hardware and software features, such as operating system, hard drive capacity or type of processor.

**You are requested to create a configuration following these recommendations, if possible (it may be impossible to simultaneously satisfy all these constraints):**

- *The final product should include an optical drive, preferably either a bluraydisc or a combined dvd rw /cd rw drive.*
- *Lowest possible price*
- *Hard drive should be of no less than 160Gb*
- *Operating System should be Windows Vista 64*
- *Intel Atom or Intel Celeron processor*
- *Lowest possible weight*
- *Include 2 or 3 Gb of memory, but preferably 2Gb. No less than 2Gb.*

## Scenario Name: EXPERIMENTAL ENVIRONMENT

### Description:

This feature model describes an experimental hardware environment, with various possible configurations of operating system and installed software.

**You are requested to create a configuration following these recommendations, if possible (it may be impossible to simultaneously satisfy all these constraints):**

- *The system should include at least one of Tomcat or Apache2 servers.*
- *Support should be provided for the highest possible number of programming languages.*
- *Operating system should be Fedora, Redhat or Ubuntu*
- *Database should be only one of CouchDB, MongoDB or Riak*
- *The maven utility should be included*