ON THE AUTOMATED ANALYSIS OF SOFTWARE PRODUCT LINES USING FEATURE MODELS

A FRAMEWORK FOR DEVELOPING AUTOMATED TOOL SUPPORT

DAVID BENAVIDES CUEVAS

EUROPEAN DOCTORAL DISSERTATION
ADVISED BY
DR. ANTONIO RUIZ-CORTÉS - DR. AMADOR DURÁN TORO

First published in June 2007 by
The Department of Computer Languages and Systems
ETSI Informática
Avda. de la Reina Mercedes s/n
Sevilla, 41012. SPAIN

Classification (ACM 1998):

Categories and subject descriptors: D.2.13 [Software Engineering]: Reusable Software: Domain Engineering; D.2.1 [Software Engineering]: Requirement / Specifications: Languages,Tools;
General Terms: Design
Additional Key Words and Phrases: software product lines, software product families, feature models, automated analysis

Support: Partially supported by the European Commission (FEDER) and Spanish Government under CICYT projects TIC2003-02737-C02-01 (AgilWeb), TIN2006-00472 (Web–Factories) and Castilla–La Mancha Government under grant PCB-02-001 (Tamansi).

Don Antonio Ruiz Cortés y Don Amador Durán Toro, profesores Titulares del Área de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla,

HACEN CONSTAR

que Don David Benavides Cuevas, Ingeniero en Informática por la Universidad de Sevilla, ha realizado bajo nuestra supervisión el trabajo de investigación titulado

On The Automated Analysis of Software Product Lines using Feature Models. A framework for developing automated tool support

Una vez revisado, autorizamos el comienzo de los trámites para su presentación como Tesis Doctoral al tribunal que ha de juzgarlo.

Fdo. Dr. Antonio Ruiz Cortés y Dr. Amador Durán Toro
Área de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
Sevilla, Mayo de 2007

Yo, David Benavides Cuevas, con DNI número 77.585.613-N,

DECLARO

Ser el autor del trabajo que se presenta en la memoria de esta tesis doctoral que tiene por título:

On The Automated Analysis of Software Product Lines using Feature Models. A framework for developing automated tool support

Lo cual firmo en Sevilla, Mayo de 2007.

Fdo. David Benavides Cuevas

In addition to the committee in charge of evaluating this dissertation and the two supervisors of the thesis, it has been reviewed by the following researchers:

- Dr. Barbara Smith (Cork Constraint Computation Centre, Ireland)
- Dr. Daniel Le Berre (Université d'Artois, France)

Universidad de Sevilla

The committee in charge of evaluating the dissertation presented by David Benavides Cuevas in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering, hereby recommends ________________________ of this dissertation and awards the author the grade ____.

____

Pere Botella López
Catedrático de Universidad
Univ. Politècnica de Catalunya

____

Miguel Toro Bonilla
Catedrático de Universidad
Univ. de Sevilla

____

Ernesto Pimentel Sánchez
Catedrático de Universidad
Univ. de Málaga

____

Frank van der Linden
Partnerships Project Manager
Philips Medical Systems

____

Vicente Pelechano Ferragud
Titular de Universidad
Univ. Politécnica de Valencia

To put record where necessary, we sign minutes in _______________, ____________________.

A mis abuelos.

Contents

List of Figures

List of Tables

Acknowledgements

If you can read this, thank a teacher.

Popular saying

During the development of your Philosophiæ Doctor Thesis you collect plenty of impressions, corrections, comments, help and support. When you finish writing your dissertation you have the time to thank all the people that have played in a way or another an important role during these years of hard work.

Many people have contributed to make this dissertation possible. However, there is no doubt that my research advisors, Dr. Antonio Ruiz and Dr. Amador Durán have put in the most effort which has materialized in my dissertation, without their help, perhaps I would have reached the end, but definitively it would have been another.

Working in a research team has been fundamental as well. I have been lucky since my thesis was developed always surrounded by a group of good research mates. Pablo Trinidad and Sergio Segura were two of the PhD students with whom I passed the most time during the development of this thesis and I would like to thank them for their patience and support. Dr. Miguel Toro and Dr. Rafael Corchuelo also helped me a lot and although I spent less time with them, they showed me their research maturity when they delighted me with their insightful comments at the beginning of my research work. Dr. José Luís Arjona, Dr. Joaquín Peña and Octavio Martín were also a great help in the development of my work for their experience and advice. Special mention goes to Dr. David Ruiz because the structure of his thesis was meticulously revised by him (among others) to be a mature framework to write research reports that inspired us in this thesis. I must also show my gratitude to the other members of the Department of Computer Language and Systems at the University of Seville for their companionship.

My work also profited enormously from colleagues at other universities and research centres. I would like to thank Professor Don Batory at the University of Texas because he pushed me to trust that my research work made a bit of sense. During my thesis I visited two research centres that I would like to remember in these acknowledgements. On the one hand, the Cork Constraint Computation Centre (4C) in Ireland was kind to receive me during three months. Special thanks to Professor Eugene Freuder, Dr. Barbara Smith, Dr. Barry O'Sullivan and Dr. Radoslaw Szymanek at 4C for their help in the field I am not an expert in, while they are. On the other hand, at the very beginning of my research, I visited the CIMAT in Mexico where Dr. Miguel A. Serrano and Dr. Miguel A. Serrano also received me very kindly. Professor Oscar Díaz and Dr. Salvador Trujillo from the ONEKIN research group in the Basque Country provided me with very "spicy" comments about my work that helped me a lot. Jean-Christophe Trigaux, Dr. Patrick Heymans and Dr. Pierre–Yves Schobbens at the FUNDP *Institut d'Informatique* and Dr. Daniel Le Berre at *Centre de Recherche en Informatique de Lens* also gave me feedback to improve my work.

Finally, and most important, I thank my parents for bringing me into this world and for supporting me unconditionally in the pursuits of my life. And to Isabel for her trustworthy love and for giving life to the most beautiful baby in the world.

Abstract

You will have only one opportunity
to make a first impression.

Popular saying

In recent years, software product lines have been introduced as a new technical paradigm to build software products focussing on the development of a set of distinctive products sharing a common part rather than building different products one by one from scratch. There is an important piece on this paradigm that serves as a model to represent the set of products of the software product line. To this end, one of the most widely used models is the so–called feature model that was proposed back in 1990 and has been a topic of research throughout recent years. Feature models have been mostly used as graphical notations to communicate different stakeholders. The automated analysis of feature models was stated as a research challenge in the original proposal, however only in recent years some publications have paid attention to this task.

In this dissertation, we present FAMA (FeAture Model Analyser), a new framework to automate the analysis of software product lines in general and feature models in particular. Its main advantages lie in its formal semantics which avoid misinterpretation, its abstraction that allows to extend the framework with other models than feature models, the capability of analysing extended feature models where features attributes are included, and finally the support of multiple solvers in the implementation of feature model analysis. Putting all this together we can set the basis to develop automated tool support.

Resumen

Sólo tendrás una oportunidad
de dar una primera impresión.

Dicho popular

En los últimos años, ha habido un gran empuje por parte de la comunidad investigadora en torno a una nueva línea de investigación. Se trata de las líneas de producto software. Las líneas de producto software se centran en proponer soluciones, métodos y técnicas para la producción y construcción de una serie de productos software que comparten características comunes pero que también tienen partes peculiares. De esta manera, la construcción de un nuevo producto se debería hacer partiendo de una base común de tal modo que no haya que empezar desde cero.

Una parte importante dentro de este nuevo paradigma de construcción de software es un modelo que represente a todos los posibles productos de una misma línea de productos. Uno de los modelos más utilizados con este fin son los modelos de características (*feature models*) que fueron propuestos por primera vez en 1990 y han sido objeto de investigación a lo largo de estos años. Hasta ahora, los modelos de características han sido principalmente usados como notaciones gráficas para comunicar a los diferentes participantes en la producción de software, desde clientes hasta desarrolladores. Es importante señalar que el análisis automático de estos modelos fue propuesto como una menta a alcanzar en el informe original de 1990, sin embargo, ha sido en los últimos años cuándo han aparecido algunas propuestas para el análisis automático de los modelos de características.

En esta tesis doctoral, se presenta FAMA (FeAture Model Analyser), un nuevo marco de trabajo para el análisis automático de líneas de producto software en general y de modelos de características en particular. Sus principales ventajas están basadas en su base formal que permite evitar interpretaciones erróneas de su semántica; su abstracción que permite extender FAMA con otros modelos distintos de los modelos de características; la posibilidad de analizar modelos de características extendidos en los que se incluyen atributos sobre las características y finalmente la inclusión de varios resolutores distintos en la implementación. De este modo, FAMA supone un paso adelante para establecer las bases que permitan la construcción de herramientas software para el análisis automático de modelos de características.

Part I
Preface

Chapter 1

Introduction

There is nothing more difficult to take in hand,
more perilous to conduct or more uncertain in its success
than to take the lead in the introduction
of a new order of things

Niccolo Machiavelli, 1469–1527
Italian dramatist, historian, and philosopher

As in other industrial environments, the way that software products are being built is changing from mass production to mass customization. Software product line engineering deals with the production of a set of products sharing a common set of features. This leads to the existence of several challenges from management to code that are being contemplated as research directions in recent years. One of the main goals of software product line engineering is the automation of tasks to produce software as automatically as possible. In fact, this has been an old dream in software engineering. There is an important piece in this puzzle which serves a model to represent the possible products of a software product line. Feature models are one of the most used models to this end and their automated analysis is still a research challenge and the main motivation of our thesis.

In this dissertation, we report on our work to design a new framework to automate the analysis of software product lines in general and feature models in particular. In this chapter, we first introduce the elements that constitute the context of our research work in Section §1.1; then, we summarise our main contributions in Section §1.2; finally, we describe the structure of the dissertation in Section §1.3.

1.1 Research context

In this section, we briefly present the main concepts we use throughout the rest of the dissertation. First we present software product lines in Section §1.1.1; we focus on feature models in section §1.1.2; finally, we outline constraint programming in Section §1.1.3

1.1.1 Software product lines

The changes in economical paradigms bring with it changes in all aspects of life. We are immersed in a new economical paradigm change whose main wave is globalization. Globalization is changing the way we *communicate* (e.g. virtual chats, forums, virtual planets such as *second life*), we *reproduce* (e.g. cloning, genetic engineering) and we *produce* (from mass production to mass customization) [30]. Obviously, software production is not excluded from this shift and consequently it is changing from single production to the production of *software product families*, a.k.a. *software product lines*.

After the industrial revolution in the late 18th century the production of goods changed due to the introduction of machines. During that time, manual and artisanal labour was replaced progressively by an industry dominated by the manufacture where machines played a very important role. In terms of production, the industrial revolution reached its zenith when the mass production of goods was converted somehow in an obvious task. *Mass production* can be defined as the production of a large amount of standardized products using standardized processes that allow to produce a big amount of the same product in a reduced time to market. Generally, in a mass production environment, the customers' requirements are the same and no customization is applied. After industrial revolution, large companies started to organize (and they are still organizing) their production in a mass production environment. A well–known example is the mass production of bicycles [69] in the National Industrial Bicycle Company of Japan. Figure §1.1 illustrates it.

However, mass production is starting to be not enough in a highly competitive and segmented market [69] and *mass customization* is due to become a must for market success. Tseng and Jiao define mass customization as "*producing goods and services to meet individual customer's needs with near mass production efficiency*" [98]. There are two key parts in this definition. First, mass customization tries to meet as many individual customer's needs as possible and second, it has to be done meeting the mass production efficiency as much as possible. Therefore, in an organization that produces its goods in a mass

Figure 1.1: *A mass of bicycles illustrating mass production.*

production environment, the production of individual products is replaced by the production of a family of similar products that allows enough flexibility to be adapted while sharing a common part. In the case of the National Industrial Bicycle Company of Japan we can imagine a web site with an ample set of possibilities to produce a bicycle taking into account individual needs. Mass customization takes these individual needs as inputs and produces customized bicycles as illustrated in Figure §1.2.

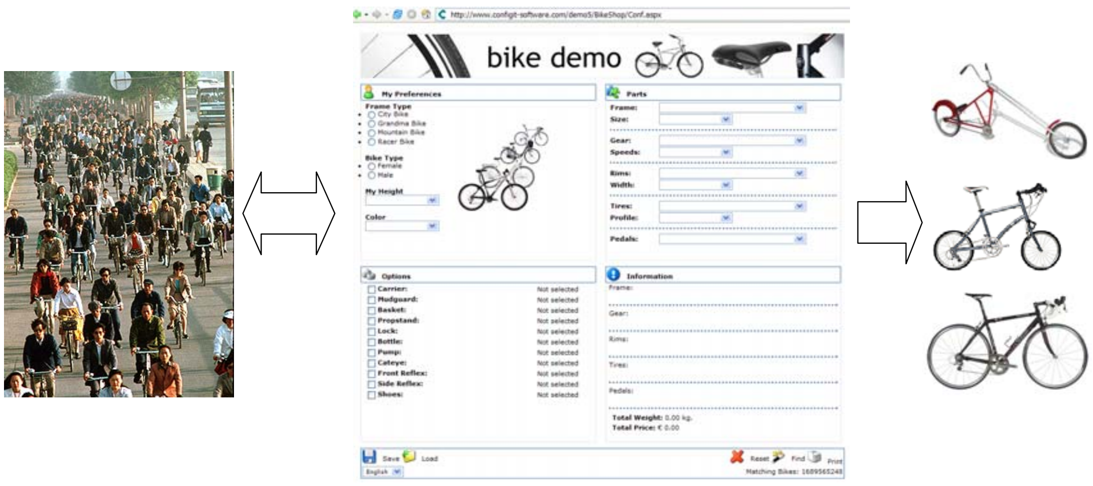


Figure 1.2: *Mass customization of bicycles.*

The software industry is a peculiar branch of industry compared to more traditional branches like bicycle production. In fact, it started to take off after the first personal computer around the mid 1970s and it is curious to remember how the sentence "*The software Industry is not industrialized*" [75] was already inscribed in the NATO conference proceedings back in 1968 and how nowadays, in many aspects, this affirmation remains valid. It is possible to make the parallelism with the history of traditional industries and we can say that the industrialization of software products started with artisanal methods, evolved to mass production and is now pursuing mass customization. The mass customization of software products is known in literature as *Software Product Lines* (SPLs) [40] or *software product families* [81].

Many software organizations start a new project from scratch. Thus, if a new project needs to be started most of the effort of previous project goes to waste. In order to achieve customer's customization, *software product line engineering* tries to avoid this situation by promoting the production of a family of software products with common features instead of producing them one by one from scratch.

According to Clements and Northrop [40], a software product line consists of a set of software products sharing a common set of features that satisfy the needs of a particular domain and that are developed from a common set of core assets in a prescribed way. Therefore, software product line engineering is about producing families of similar systems rather than the production of individual systems. Software product line engineering consists of three main activities: *domain engineering* (also called *core asset development*) and *application engineering*(also called *product development*) and *management*. These three activities are complementary and provide feedback to each other. Domain engineering deals with the production of software assets to be used in different products of the product line. On the other hand, application engineering deals with the production of individual systems from core assets and individual needs and management that is responsible for giving resources, coordinating, and supervising domain and application engineering activities.

1.1.2 Feature models

The essence of software product lines is the systematic and efficient creation of different products to satisfy customer's needs. A key technical question that confronts software engineers is how to specify a particular product in a software product line. When this question was first considered, there was ample evidence for a solution. People were familiar with automobile product lines: cars offered optional features to satisfy different customer needs. Product lines of televisions, appliances, and plumbing fixtures were also familiar, all offering a host of optional features for similar reasons. The solution was simple: products of a product line were differentiated by their *features*, where a *feature* is an increment in product functionality. While single software systems are specified in terms of features, software product lines are specified using *feature models*.

Feature models are recognized in the literature to be one of the most important contributions to software product line engineering [11, 43]. A key task in software product line engineering is to capture commonalities and variabilities among products [71] and feature models are used for this purpose.

A feature model represents all possible products of a software product line. It is a hierarchically arranged set of features composed by:

i. relationships between a parent (or compound) feature and its child features (or subfeatures).

ii. cross–tree constraints that are typically inclusion or exclusion statements of the form: *if feature F is included, then features A and B must also be included (or excluded).*

Since feature models were first presented in 1990 [65], there have been many publications and proposals to extend, improve and modify the original feature model. However, despite years of research, there is no consensus on a feature model notation. There are two branches of notations: cardinality–based feature models [46] and feature models without cardinalities [9]. The main distinction is that the former allows more complex relations in the tree structure than the latter.

Right from the introduction of feature models an error-prone task was presented that still has not been fully solved: the *automated analysis of feature models*[11]. The analysis of a feature model consists of the observation of its properties such as whether a feature model is void (it represents no products), whether a feature model has internal *dead features* (features that although represented in a feature model are not in any of its represented products), determining the number of products represented by a feature model and so forth. Feature models are used as input in many other software product line engineering processes such as code generation [9, 12], requirements engineering [81, 86] or feature oriented model driven development [95]. Therefore, the analysis of feature models is an important task since it must be done before starting any other activity.

1.1.3 Constraint programming

Constraint programming [97] has been a topic of research in artificial intelligence for the last few decades and is recognized to be one of the strategic research directions in computer research by the ACM (*Association for Computing Machinery*) since 1996 [61].

In the words of Freuder [56] "*constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it*".

One of the main branches of research in constraint programming is *constraint satisfaction problems* (CSPs). A CSP is defined as a set of variables, each ranging on a finite domain, and a set of constraints restricting all the values that variables can take simultaneously. A solution to a CSP is the assignment of a value from its domain to every variable, in such a way that all constraints are satisfied simultaneously.

1.2 Contributions

1.2.1 Summary of contributions

In this section we summarize the main contributions of our dissertation and research work.

On the one hand, our dissertation focuses on the automated analysis of feature models in the context of software product lines. We have analysed current proposals to automate the analysis of feature models. We concluded that there are some gaps that have to be bridged. We have devised a rigorous framework called FAMA (FeAture Model Analyser) to solve these problems. FAMA is designed on four levels from the more abstract to the more detailed. The contributions of this dissertation are summarized as follows:

i. FAMA is defined using Z formal language which provides a high level of rigour in the definitions avoiding misinterpretations.

ii. FAMA defines analysis operations at a higher level of abstraction which allows to reuse semantics in more detailed levels and allows to analyse software product lines using other models than feature models.

iii. In addition to basic feature models, FAMA supports the analysis of feature models with attributes.

iv. In the lower level of FAMA, it allows to implement the automated analysis of feature model using different solvers. It is important to remark that FAMA uses general CSP solvers which is a novel contribution.

The aforementioned contributions have been partially published in journals and papers, as described afterwards. However, the framework as a whole has not been published yet. These results are part of an ongoing paper to be sent to a journal in the early future.

On the other hand, during our research work and before writing this dissertation we have made some contributions that can be summarized as follows:

i. **OPERATIONAL SUPPORT FOR FEATURE MODEL ANALYSIS**

   Problem statement: Feature models need an operational support to automate its analysis.

   Contribution: We have pioneered the mapping of extended feature models to constraint programming to provide an operational support. In addition, we have proposed to use a multi solver approach to analyse feature models.

ii. **EXTENDED FEATURE MODELS**

   Problem statement: Basic feature models have limited expressiveness.

   Contribution: We have proposed to add attributes and relationships between them to features, i.e. we have proposed the notion of extended feature models to provide more expressiveness to basic feature models. In addition, we have provided a first approach to automate the analysis of extended feature models.

iii. **TOOL SUPPORT FOR FEATURE MODEL ANALYSIS**

   Problem statement: The analysis of feature models needs tool support.

   Contribution: We have developed a set of techniques and tools for supporting automated analysis of feature models. As a result, we have built a first version of a FAMA Eclipse plug–in.

iv. **SURVEY AND RESEARCH AGENDA**

Problem statement: The automated analysis of feature models is a new research area that requires a survey and research agenda to follow.

Contribution: We have surveyed the state of the art on the automated analysis of feature models and provided a research agenda to follow in the future.

v. **WEB SERVICES AND SOFTWARE PRODUCT LINES DEVELOPMENT**

Problem statement: Web services need a development process to achieve service oriented mass customization.

Contribution: We have proposed to gather web service development and software product lines as the development process of future service oriented applications.

vi. **INDUSTRIAL EXPERIENCES**

Problem statement: Industry generates research problems while academia provides solutions to be implemented in industry.

Contribution: We explored topics of research in industry and we have applied some of our solutions in industrial settings.

vii. **FEATURE MODEL ERROR ANALYSIS**

Problem statement: Error analysis on feature models needs an automated platform.

Contribution: We layed the foundations on the use of theory of diagnosis for error analysis in feature models. This is a contribution that follows up the results of this dissertation.

Table §1.1 summarizes our main contributions. The first column corresponds to the category of the contribution. The second column contains acronyms of the publications. The third column is the number of citations to our papers. Our main publications are highlighted in boldface. Acronyms will be better understood by reading the next Section.

1.2.2 Publications in chronological order

Our research on software product lines started in a large software company (Telvent) with whom we participated in two European projects: CAFE

Contributions	Publications	Citations
Operational support	ICSR04, ADIS04, SVM04, **CAiSE05**, SEKE05	24
Extended feature models	ECOOP03	-
Tool support	SPLC05, GTTSE05, SPLC06, VAMOS07	1
Survey and research agenda	JISBD06, **CACM06**, **COMPUTER07**	5
WS–SPL	SIT02, ZOCO02, IICS05	1
Industrial experiences	PFE04, **TELVENT07**	-
Error analysis	CAiSE06, APLE06, **JSS07**	-
Total	**19** + 2 (submitted)	**31**

Table 1.1: *Summary of publications grouped by subject.*

and FAMILIES [99]. Our experiences in those projects allowed us to start establishing contacts and topics of research for our PhD dissertation.

Our research work has followed a clear path, allowing us to publish our contributions in mainstream conferences and journals. Below is a complete list of publications in which we point out the main cornerstones of the development of our results in chronological order .

- Our first result was published in a national conference [13]. We proposed a set of techniques using XML to represent the variability of a software product line and claimed the need of including extra–functional aspects in software product line models. This paper was prepared in conjunction with Dr. M.A. Serrano and Dr. C. Montes de Oca from CIMAT at Mexico.

  SIT02. D. Benavides, A. Durán, M.A. Serrano, and C. Montes de Oca. Quality Of Service in System Families Based on Web Services.In *Proceedings of the Symposium on Informatics and Telecommunications*, Seville, Spain, September, 2002.

- Secondly, in [18] we provided an overview of the benefits that a software product line approach could bring to the development of applications based on web services. This paper was extended and accepted in [19].

ZOCO02. D. Benavides, A. Ruiz-Cortés, O. Martín, and J. Bermejo. A first approach to build product lines of MOWS. In *Proceedings of the ZOCO02 workshop*, El Escorial, Madrid, 2002.

IICS05. D. Benavides, A. Ruiz-Cortés, M.A. Serrano, and C. Montes de Oca. A first approach to build product lines of multi-organizational web based systems (MOWS). In *Innovative Internet Community Systems, IICS 2004*, volume 3473 of *Lecture Notes in Computer Sciences*, pages 91–98. Springer–Verlag, 2005.

- Later, in [15] we focused our challenges on introducing extra–functional information in software product lines. We can say that the attendance to this workshop provided us the necessary background to start working on feature models which are one of the pillars of strength of this dissertation.

ECOOP03. D. Benavides, A. Ruiz-Cortés, R. Corchuelo, and A. Durán. Seeking for extra-functional variability. In *Proceedings of the ECOOP Workshop on Modeling Variability for Object-Oriented Product Lines*, Darmstadt, Germany, 2003.

- The study of feature models allowed us to present a contribution to the software product line conference [53]. This can be seen as a marginal result but it is important to remark that it was presented in the major research forum of software product lines and it was the result of joint work with a large company.

PFE04. A. Durán, D. Benavides, and J. Bermejo. Applying system families concepts to requirements engineering process definition. In *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Sciences (LNCS)*, pages 140–151. Springer-Verlag, 2004.

- After a deep study of feature models, we concluded that an automated support to the analysis of feature models was mandatory. We presented our first approach in [16] and later in [22] where we provided a method to translate a feature model into a constraint satisfaction problem. This allows us to have an automated support for feature models analysis:

ICSR04. D. Benavides, A. Ruiz-Cortés, R. Corchuelo, and O. Martín-Díaz. SPL needs an automatic holistic model for software reasoning with feature models. In *International Workshop on Requirements Reuse in System Family Engineering*, pages 27–33, Madrid, Spain, 2004. Universidad Politécnica de Madrid.

SVM04. D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Coping with automatic reasoning on software product lines. In *Proceedings of the 2nd Groningen Workshop on Software Variability Management*, November 2004.

- We presented a paper in a national workshop where we proposed to use the automated analysis of feature models for the management of production plans [92]:

ADIS04. P. Trinidad, D. Benavides, and A. Ruiz-Cortés. Improving decision making in software product lines product plan management. In J. Dolado, I. Ramos, and J. Cuadrado-Gallego, editors, *Proceedings of the V ADIS 2004 Workshop on Decision Support in Software Engineering*, volume 120. CEUR Workshop Proceedings (CEUR-WS.org), 2004.

- The former papers were the starting point for our paper in the CAiSE conference where we presented a mapping from feature models to constraint satisfaction problems. [23].

CAiSE05. D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, volume 3520 of *Lecture Notes in Computer Sciences*, pages 491–503. Springer–Verlag, 2005.

- An extension of [23] was accepted in [24].

SEKE05. D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Using constraint programming to reason on feature models. In *The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE*, 2005.

- Jointly with the ONEKIN group at the University of the Basque Country was materialized as a workshop paper [29] within the Software Product Lines Conference (SPLC'05). We proposed the modularization of feature models using XML Schemas.

SPLC05. D. Benavides, S. Trujillo, and P. Trinidad. On the modularization of feature models. In *First European Workshop on Model Transformation. Software Product Line Conference*, September 2005.

- We participated in the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05) where we submitted an abstract of our work that was selected with other fourteen other submissions to be presented during the summer school. Later, the organizers selected six presentations to be written for inclusion in a chapter of a special volume of Springer–Verlag Lecture Notes in Computer Science where it was published [27]. We compared two CSP solvers on the automated analysis of cardinality–based feature models:

  GTTSE05. D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, chapter Using Java CSP Solvers in the Automated Analyses of Feature Models, pages 389–398. Springer–Verlag, 2006.

- We published some papers in different workshops at SPLC'06 [26, 94] and CAiSE'06 [93] where we presented the first steps towards a framework for the automated analysis and how to automatically deal with errors in feature models:

  SPLC06. D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Proceedings of the Workshop on Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–45, 2006.

  APLE06. P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and M. Toro. Explanations for agile feature models. In X. Franch and K. Cooper, editors, *1st International Workshop on Agile Product Line Engineering (APLE'06)*, 2006.

  CAiSE06. P. Trinidad, D. Benavides, and A. Ruiz-Cortés. A first step detecting inconsistencies in feature models. In *CAiSE Short Paper Proceedings*, 2006.

- In 2006, we presented a paper at the Spanish conference *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)* [25] where we surveyed the state of the art on the automated analysis of feature models:

  JISBD06. D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 367–376, 2006.

- We have published a paper in a special issue on software product lines of the "Communications of the ACM" journal [11] jointly with professor

Don Batory, head of the software product lines group at the University of Texas. We presented the main research challenges we envision for the future in the automated analysis of feature models:

CACM06. D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47, December 2006.

- We presented a first implementation of our framework in [28]:

VAMOS07. D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.

- We have submitted a paper to the Journal of System and Software where we present a first extension of the framework presented in this dissertation to cope with the operations of detection and explanation of errors in feature models [91]:

JSS07. P. Trinidad, D. Benavides, A. Ruiz-Cortés, A. Durán and M. Toro. Agile error analysis of feature models. *Journal of System and Software (conditionally accepted)*, 2007.

- We have contributed to a book on experiences in software product lines in action jointly with a large company [31]:

TELVENT07. J. Bermejo, P. Trinidad, D. Benavides, and A. Ruiz-Cortés. *Software product lines in action*, chapter Experience reports: Telvent. Springer, June 2007.

- We have submitted a paper to the journal "IEEE Computer" by invitation. In this paper we introduce software product lines, revise the state of the art on the automated analysis of feature models and provide a revised research agenda [14]. This is also a joint work with professor Batory:

COMPUTER07. D. Benavides, A. Ruiz-Cortés, and D. Batory. Automated analysis of software product lines using feature models: Applications, current solutions and challenges. *IEEE Computer (submitted)*, 2007.

Table §1.2 summarises and classifies our contributions according to the place of publication.

Category	Number	Acronyms
Published by LNCS	3	IICS05, PFE04, **CAiSE05**
Book chapters	2	GTTSE05, **TELVENT07**
International conferences	1	SEKE05
International Workshops	8	ECOOP03, ICSR04, SVM04, SPLC05, SPLC06, APLE06, CAiSE06, VA-MOS07
National conferences	2	SIT02, JISBD06
National workshops	2	ADIS04, ZOCO04
Journals	1 + 2 (submitted)	**CACM06**, **JSS07**, **COM-PUTER07**
Total	19 + 2	

Table 1.2: *Summary of publications grouped by place of publication.*

1.2.3 Citations

Our work has been cited in the context of software product lines and feature modelling. The paper that has received most quotes has been the one that we consider as our seminal paper [23]. It may be important to highlight that our paper is the most quoted paper of CAISE'05 according to google scholar (http://scholar.google.com). Below we enumerate the papers that have cited our work. In parenthesis we indicate the acronyms of the papers cited:

JCR Journals

i. P. Schobbens, J.C. Trigaux P. Heymans, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb. 2007. [87]. (CAiSE05).

ii. J. Pena, M. Hinchey, and A. Ruiz-Cortés. Multiagent system product lines: Challenges and benefits. *Communications of the ACM*, 49(12):82–84, December 2006. [78]. (CAiSE05,IICS05).

Other Journals

iii. M. Cengarle, P. Graubmann, and S. Wagner. Semantics of UML 2.0 interactions with variabilities. *Electronic Notes in Theoretical Computer Science*, 160:141–155, 2006. [38]. (CAiSE05).

iv. W. Zhang, H. Mei, and H. Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*, 11(3):205–220, June 2006. [108]. (CAiSE05).

Conferences indexed by ISI

v. D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer–Verlag, 2005. [9]. (CAiSE05).

vi. T. Asikainen, T. Mannisto, and T. Soininen. A unified conceptual foundation for feature modelling. In *Software Product Line Conference, 2006 10th International Conference*, pages 31–40. IEEE Press, 2006. [6]. (CAiSE05).

vii. J. White, D.C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating product-line variant selection for mobile devices. In *Proceedings of the 11th Annual Software Product Line Conference (to appear)*, 2007. [106]. (CAiSE05)

viii. V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 201–210, New York, USA, 2006. ACM Press. [3]. (CAiSE05).

ix. P. Schobbens, P. Heymans, and J. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, 2006.[86]. (CAiSE05).

x. A. Metzger, P. Heymans, K. Pohl, and P. Y. Schobbens. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (to appear)*, 2007. [76]. (CAiSE05, JISBD06, CACM06).

xi. D. Wagelaar and V. Jonckers. Explicit platform models for MDA. In *MODELS 2005 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005. [104]. (CAiSE05).

xii. J. Peña, M. Hinchey, A. Ruiz-Cortés, and P. Trinidad. Building the core architecture of a multiagent system product line: With an example from a future nasa mission. In *7th International Workshop on Agent Oriented Software Engineering*, Lecture Notes in Computer Sciences. Springer–Verlag, 2006. [79]. (CAiSE05, JISBD06).

xiii. O. Diaz, S. Trujillo, and S. Perez. Turning portlets into services: The consumer profile. In *Proceedings of the 16th International World Wide Web Conference*, 2007. [52]. (SPLC05).

International workshops

xiv. K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA 2005*, 2005. [48]. (CAiSE05).

xv. J. Trigaux, P. Heymans, P. Schobbens, and A. Classen. Comparative semantics of feature diagrams: FFD vs. vDFD. In *Proceedings of the Fourth International Workshop on Comparative Evolution in Requirements Engineering (CERE)*, 2006. [90]. (CAiSE05).

xvi. P. Heymans, P.Y. Schobbens, J.C. Trigaux, R. Matulevicius, A. Classen, and Y. Bontemps. Towards the comparative evaluation of feature diagram languages. In *Proceedings of the Software and Services Variability Management Workshop - Concepts, Models and Tools*, 2007. [62]. (CAiSE05, JISBD06).

xvii. R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *Proceedings of the ACM SIGSOFT First Alloy Workshop*, pages 71–80, 2006.[57]. (CAiSE05)

xviii. D. Wagelaar. Towards context-aware feature modelling using ontologies. In *Proceedings of the MoDELS 2005 International workshop on MDD for Software Product Lines*, 2005. [103]. (CAiSE05).

xix. L. Etxeberria, G. Sagardui, and L. Belategi. Modelling variation in quality attributes. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007. [54]. (SEKE05, CAiSE05, SPLC06).

National conferences

xx. G. Aldekoa, S. Trujillo, G. Sagardui, and O. Díaz. Experience measuring maintenability in software product lines. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006. [2]. (CAiSE05).

Books and PhD dissertations

xxi. V. Pankratius. *Software product lines for digital information products*. Karlsruhe University Press, 2007. [77]. (CACM06)

xxii. S. Trujillo. *Feature Oriented Model Driven Product Lines*. Doctor of philosophy, University of the Basque Country, San Sebastián, Spain, 2007. [95]. (CAiSE05, SPLC05, VAMOS07).

	Number
JCR journals	2
Other journals	2
Conferences indexed by ISI	9
International workshops	6
National conferences	1
Books and dissertations	2
Recommended bibliography	4
Total	**26**

Table 1.3: *Summary of citations.*

Recommended bibliography

xxiii. "*Feature Oriented Programming*" post–graduate course of the Computer Science Department of the University of Texas at Austin (USA). †1 (CAiSE05).

xxiv. "*Tutorial on Feature Oriented Programming*". Held in Braga (Portugal) during the Summer School on Generative and Transformational Techniques in Software Engineering 2005. †2 (CAiSE05).

xxv. "*Tutorial on Feature Modularity for Product–Lines*". Held in Portland (USA) during the Generative Programming and Component Engineering Conference 2006. †3 (CAiSE05).

xxvi. "*From SAT to SAT4J. Providing efficient SAT solvers for the Java platform*". Presentation of SAT4J to the constraints and proofs group at Polytech'Nice–Sophia. (CAiSE05). †4

†1 `www.cs.utexas.edu/users/dsb/FOPCourse.html`

†2 `www.di.uminho.pt/GTTSE2005`

†3 `www.gpce.org`

†4 `www.sat4j.org/sat4joverview.pdf`

Table §1.3 summarises the type of citations we have received.

1.2.4 Research visits

During the development of this thesis, we have visited two research centres:

- ***Centro de Investigación en Matemáticas* (CIMAT)**[†5]: This research centre located in Guanajuato (Mexico) is considered to be one of the most important research centres in Latin America in the field of mathematics. They are a small group of researchers interested in software engineering. The hosts were Dr. Carlos Montes de Oca and Dr. Miguel A. Serrano. During the summer of 2001, in the early stages of our research, we spent one month in this centre. The main topic of research was the quality of service variability in multiorganizational web–based systems in the context of software product lines. The results of the visit were materialized in two seminars held at CIMAT with an audience of more than one hundred people. In addition we have written two joint papers that have been presented in a national [13] and an international conference[19].
- **Cork Constraint Computation Centre (4C)**[†6]: This research centre located in Cork (Ireland) is one of the most important research centres in the field of constraint programming in the world according to its publications and staff. In 2005, we spent three months in 4C where we improved our knowledge on constraint programming. The hosts were Dr. Barbara Smith and Prof. Eugene Freuder. The results of the visit were materialized in a framework agreement between the University of Seville and 4C that opened the doors to the visit of Pablo Trinidad later in 2006. In addition, we are working on a joint paper to analyse the complexity of some operations on feature models [20].

1.3 Structure of this dissertation

This dissertation is organized as follows:

Part I: Preface. It comprises this introduction only.

[†5]`www.cimat.mx`
[†6]`www.4c.ucc.ie`

Part II: Background information. Here, our goal is to provide the reader with a deep understanding of the research context in which our work has been developed. In Chapter §2, we survey the most common notations of feature models providing some examples. In Chapter §3, we list the operations we have identified in the automated analysis of feature models and we analyse current solutions. In Chapter §4, we present constraint programming in a nutshell providing a formalization of constraint satisfaction problems using Z.

Part III: Our Contribution. This part is the core of our dissertation, and is organized in five chapters. In Chapter §5, we motivate our research and explain why current solutions do not cover all the properties we have identified. In Chapter §6, we rigorously present the FAMA framework and we describe the abstract foundation layer of FAMA. In Chapter §7, we describe the characteristic model layer of our framework where we provide semantics to feature models using Z specification language. In Chapter §8, we describe the operational paradigm layer where feature models are expressed in terms of constraints but without being coupled to any constraint solver implementation. Finally, in Chapter §9, we present a proof of concepts implementation of FAMA using concrete solvers.

Part IV: Final Remarks. It consists of Chapter §10 in which we report our main conclusions and future research.

Part V: Appendices. The implementation of an Eclipse plug–in for the automated analysis of feature model is described in Appendix §A, some mathematical notes are presented in Appendix §B and the acronyms used throughout the thesis are presented in Appendix §C.

Part II
Background information

Chapter 2

Feature models

First learn the meaning of what you say, and then speak.

Epictetus, 54–135 AD
Roman (Greek-born) slave & Stoic philosopher

In this chapter, we focus on feature models describing the most common notations of feature models and providing some examples. In Section §2.1, we introduce feature models; In Section §2.2, we describe basic feature models; In Section §2.3, we describe cardinality–based feature models; In Section §2.4, we describe extended feature models providing some examples of basic, cardinality–based and extended feature models in Section §2.5. Finally in Section §2.6 we summarise the chapter and detail our main contributions in this field.

2.1 Introduction

Software product lines are conceived for the efficient and systematic creation of software products. This aim could not be achieved if software reuse is not planned and controlled. A product that is a part of a software product line is created based on the reuse of existing assets. Some of these assets are common to other products in the same product line and some others are specific for individual products. Therefore, a fundamental artifact in this context is a model to capture and express variabilities and commonalities among different products. A key technical question that confronts software engineering when coping with software product lines is how to specify a particular product and how to specify the software product line itself to express variabilities and commonalities.

The concept of product lines is not new in other engineering and manufactory branches. Car product lines, appliance product lines, computers product lines and even hamburgers product lines are commonly applied in practice. In all these product lines the way a product is specified is similar: using product features. Then, in software product lines, features are also a widely used way to specify individual products and feature models are used to specify the software product line itself where a *feature* is considered to be an increment in product functionality [11, 65].

Feature models are considered by some authors as one of the most important advances in software product line engineering [11, 43, 67], because it is more natural and intuitive for both customers and developers to express commonalities and variabilities of a software product line in terms of features since features are understood by all stakeholders [67].

A feature model represents all possible products of a software product line in a single model. Feature models are used in different scenarios such as requirements engineering [64, 73, 108], architecture definition [79], architecture maintainability measurement [2] code generation [9, 12, 43, 82] or portlet–based applications [96].

A feature model is a hierarchically arranged set of features and consists of:

i. Relationships between a parent feature and its child features.

ii. Cross–tree constraints that are typically inclusion or exclusion statements of the form *"if feature F is included, then feature X must also be included (or excluded)"*.

In 1990, Kang *et al.* [65] proposed feature models for the first time. However, despite years of research, there is no consensus on the modeling artifacts allowed in feature models and many extensions have been proposed since then. Most of the extensions are based on the relationships allowed between parent and child features.

First, the original feature model notation called *FODA* [65] was proposed. Later, *Feature-RSEB* [58] was presented as a FODA extension with an additional relationship. Finally, Riebisch *et al.* [84] and Czarnecki *et al.* [42, 46] proposed *cardinality–based feature models* where cardinalities were introduced. In the following sections, we summarize these notations.

2.2 Basic feature models

There are two notations that we have classified as basic feature models. FODA feature models and feature–RSEB feature models.

2.2.1 FODA feature models

The modeling elements of a feature model are features and relationships between parent and child features. FODA defined three different kind of relationships [†1]:

- **Mandatory.** There exist a *mandatory* relationship between a parent and a child feature when the child feature of the relationship is required to be included in a product when the parent feature is included. For instance, it is *mandatory* to have the body of a car in the feature model of Figure §2.7 (pag. 33).

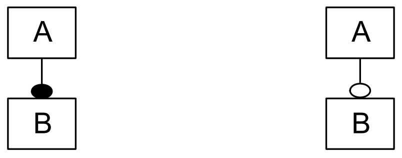


Figure 2.1: mandatory *and* optional *relationships.*

[†1]for some samples see Section §2.5 (pag. 32)

- **Optional.** There exist an *optional* relationship between a parent and a child feature when the child feature can be included in a product or not when the parent feature is included. For instance, it is *optional* to have air conditioning in a car.

- **Alternative.** There exist an *alternative* relationship between a parent feature and a set of children features when only one child feature can be included in a product when the parent feature is included. For instance the transmission of a car can only be automatic or manual.

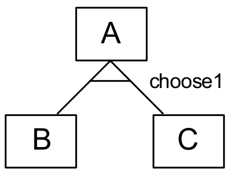


Figure 2.2: *An* alternative *relationship between features* A, B *and* C.

In addition, cross-tree constraints are allowed that restrict feature combinations. These cross–tree constraints allowed in FODA are:

- Excludes. A feature X excludes Y means that if X is included in a product, then Y should not be included and backwards. For instance, PDA excludes Repository in the feature model of Figure §2.8 (pag. 34).

- Requires. A feature X requires Y means that if X is included in a product, then Y should be included as well, but not backwards. For instance, CongressManagement requires Repository in the feature model of Figure §2.8.

2.2.2 Feature–RSEB feature models

In 1998, Griss *et al.* [58] presented an extension of FODA feature models that is usually referred as Feature–RSEB. Although they presented a new graphical notation, the semantic of the model was very similar, because the relationships allowed for features were the same as the original FODA model plus an additional relationship, namely: *mandatory*, *optional*, *alternative* and cross–tree constraints: *excludes* and *requires*. In addition, a new relationship was introduced:

- Or: There exist an *or* relationship between a parent feature and a set of children features when one or more child features can be included in a product when the parent feature is included. For instance, the Engine of a car can be Gasoline or Electric or both.

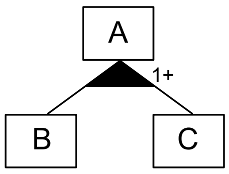


Figure 2.3: *An* or *relationship between features* A*,* B *and* C.

2.3 Cardinality–based feature models

Another set of feature models are those based on *cardinalities* (also known as *multiplicities*). The main motivation to extend feature models with cardinalities was that some cases could not be modeled using *alternative* and *or* relationships [83].

First, some authors [42, 83, 84] proposed the extension of feature models with multiplicities. *Mandatory* and *optional* relationships kept the same meaning as in the original FODA models, meanwhile *alternative* and *or* relationships were generalized in such a way that only a *set* relationship was considered:

- Set. A set of child features are said to have a *set* relationship if a number of features can be included in a product when their parent feature is included. This number depends on the cardinality. Thus, an *alternative* relationship is equivalent to a *set* relationship with cardinalities $\langle 1-1 \rangle$ (e.g. the relationship R6 of Figure §2.8 on page 34 can be expressed using an *alternative* relationship or a *set* relationship with cardinality $\langle 1-1 \rangle$). Likewise, an *or* relationship is equivalent to a *multiplicity* relationship with cardinalities $\langle 1-N \rangle$ being N the number of features in the relationship (e.g. the relationship R8 of Figure §2.8 can be expressed using an *or* relationship or a *set* relationship with cardinality $\langle 1-4 \rangle$)

Later, Czarnecki *et al.* [44, 46] proposed cardinality–based feature models. Their main motivation was driven by practical applications [42] and “conceptual completeness”. The major change in the semantics of cardinality–based

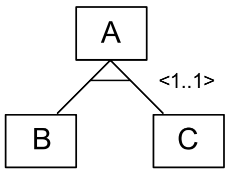


Figure 2.4: *A* set *relationship between features* A, B *and* C.

feature models regarding multiplicity–based notation is the introduction of a new relationship:

- Feature cardinality. With the introduction of feature cardinality, *optional* and *mandatory* relationships are generalized in the sense that the times that a feature is included in a product is determined by its cardinality. Thus, an *optional* relationship is equivalent to a feature *cardinality* relationship with cardinality [0..1] and a *mandatory* relationship is equivalent to a feature *cardinality* relationship with cardinalities [1..1].

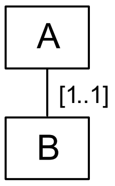


Figure 2.5: *A* cardinality *relationship between features* A *and* B.

Figure §2.6 presents a summary of the notations and the equivalences between them [†2].

2.4 Extended feature models

Basic feature models are suitable to express commonality and variability among different products in a software product line. However, sometimes it is necessary to extend feature models to include more information about features

[†2]The graphical representation of different proposals vary from one to another. Thus, for the sake of simplicity we decided to use the same graphical notation

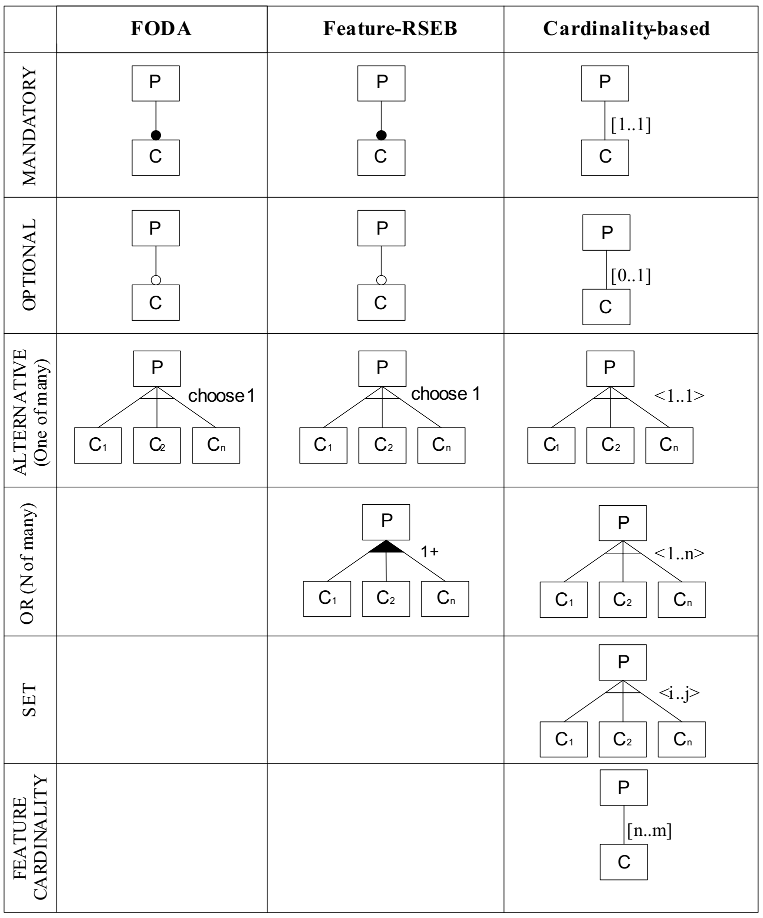


Figure 2.6: *Comparison of feature model notations.*

like the inclusion of attributes in feature models. These type of models where additional information is included are called *extended feature models*[11, 23].

In fact, FODA already contemplated the inclusion of some additional information in feature models [65, pag. 34], for instance, relationships between features and features attributes were introduced. Later, in 1998 Kang *et al.* [66] made an explicit reference to what they called 'non–functional' features related to features attributes. In 2001 Chastek *et al.* [39], proposed some guidelines for feature modelling: in [39, pag. 19], the authors once again made the distinction between functional and quality features and pointed out the need of a specific method to include attributes in feature models. In addition, other group of authors have also proposed the inclusion of attributes in feature models [42, 44, 46–48, 54, 88] and we have also contributed to this discussion [13, 15, 16, 22–24].

Despite of the consensus about the inclusion of attributes and attributes relationships in feature models, there is not a standard notation nor an agreement on the type of attributes that can be included in feature models.

The following concepts are usually used when dealing with feature attributes:

- Feature. A prominent characteristic of a product or an increment in product functionality.
- Attribute. An attribute of a feature is any characteristic of a feature that can be measured. For instance, the PHP version of a module in the example in Figure §2.8, its development cost, the number of lines of code, and so forth.
- Attribute domain. It specifies the space of possible values for an attribute. Every attribute has an associated domain. It is possible to have discrete domains (e.g. integers, booleans, enumerated values) or continuous domains (e.g. real).
- Attribute value: The value of an attribute belonging to the domain. There can be a default value in the case the feature is not selected. A value can be directly a value on the domain (basic attributes) or an expression combining other attributes of the same or other features (derived attributes). For instance, the PHP version of a module can be a basic attribute defined by $PHPversion = 5$. But the PHP version of a complete application that has several PHP modules can be a derived attribute defined as the maximum of the versions of the different modules (see Figure §2.9 for an example).
- Attribute relationships. A relationship between one or more attributes of a feature or a feature itself (e.g. in the feature model of Figure §2.8 "WSInterface requires Modules.version $>$ 4").

2.5 Some examples

In this Section we present some basic examples of feature models. Our intention is to provide an overview of small feature models that are commonly found in research papers in order to illustrate the operations we describe in subsequent Chapters.

2.5.1 Car feature model

Software is a key part of today's automobiles. Software monitors the correct operation of a car (oil pressure, fuel consumption, etc.) and improves the car's performance (e.g., adjusting fuel intake for better efficiency). The software embedded into a car is determined by the car's features; each distinct set of features defines a unique program in a software product line. Suppose we only consider the features of transmission type (automatic or manual) and engine type (electric or gasoline). Figure §2.7 depicts a feature model of this product line using basic feature model notation. A car consists of a body, transmission, engine, and optionally a cruise control. A transmission is either automatic or manual (choose one), and an engine is electric–powered, gasoline–powered, or both.

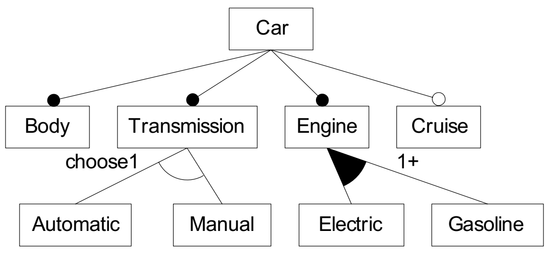


Figure 2.7: *Car feature model.*

2.5.2 James feature model

A software product line of collaborative systems on the web can be conceived. Different modules can be added which correspond to different products in the product line.

Figure §2.8 depicts the feature model of JAMES [55] using cardinality–based notation. JAMES is a framework for developing web collaborative systems. Every JAMES product has to have the Core elements which are the base of the product to be operative. In addition, there are JAMES products with Web Services Interface (WSInterface) management and other without it. Every JAMES product can have data base (DB), or LDAP user authentication, but only one. In a JAMES product there are products with PC or PDA graphical user interface, or both at the same time (because the cardinality). Finally, feature CongressManagement requires Repository and feature Repository excludes PDA, i.e. it is no possible to have a JAMES product with both features at the same time.

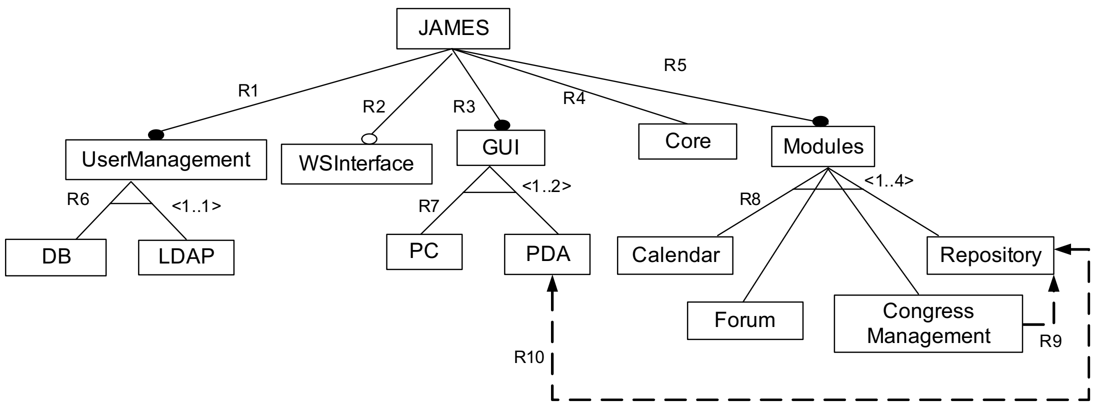


Figure 2.8: *James feature model.*

2.5.3 Extended feature model

There is not a widespread notation for extended feature models and it is out of the scope of this dissertation to propose one. In this Section we simply propose a possible notation but not with the aim of establishing it as a standard but to illustrate a possible example of extended feature models.

To extend the basic notations of feature models, we can use a box as used in UML classes where the upper part represents the name of the feature and the bottom part allows defining attributes for the feature.

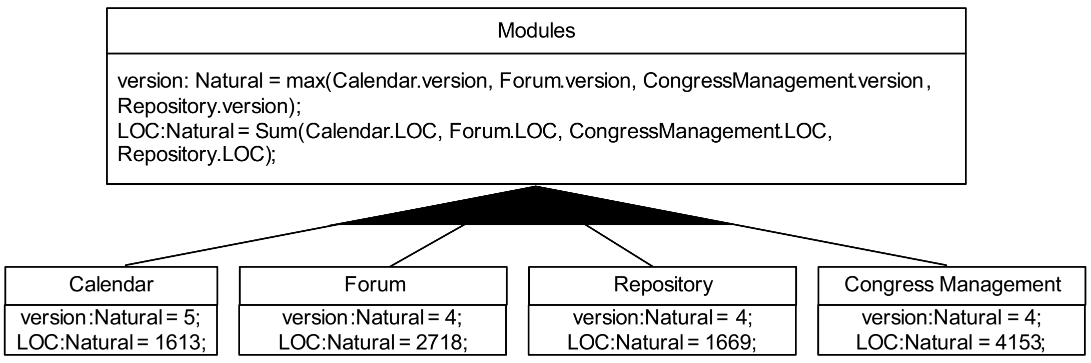


Figure 2.9: *Extended James feature model example.*

An example of representing attributes in feature models is illustrated in Figure §2.9. Any of the child features of feature Modules can have an attribute named version referred to the versions of PHP (JAMES is implemented using PHP) that the module requires to work (natural domain). These are examples of basic attributes. Similarly the attribute LOC is introduced to represent the number of lines of code of the module.

An example of derived attribute would be the attribute *version* of the feature Modules because it depends on the version of the modules selected in every product. In this case the version required for modules to run will be the maximum of the versions of the child features of Modules. It would depend on the type of relationships and the type of attributes how the derived attributes would be made up.

2.6 Summary

In this chapter we have summarised the most common feature model notations divided into three main groups, namely, basic, cardinality–based and extended feature models. We have also presented some simple examples to guide the comprehension of the different notations. We published part of this chapter in the *Jornadas de Ingeniería del Software y Bases de Datos* 2006 [25].

Chapter 3

Automated analysis of feature models

Computers are useless.
They can only give you answers.

Pablo Picasso, 1881–1973
Spanish Cubist painter

We provide in this chapter with a description of the operations that we have identified on the automated analysis of feature models surveying the main proposals in the literature. In section §3.1, we first introduce the concept of automated analysis of feature models. We describe the operations of analysis in Section §3.2 and the operations of modification in Section §3.3. In Section §3.4, we survey the main proposals to automate the analysis of feature models and we finally summarise the chapter and present our main contributions in Section §3.5.

3.1 Introduction

In 1990 feature models were proposed as a way of specification of software product lines by Kang *et al.* [65]. Managing large feature models was already described as a difficult and error–prone task. In fact, in the original FODA report, automated tool support for the *analysis of feature models* was already pointed as an important challenge and a prototype tool was described [65, pag. 70].

Feature models are being used in many activities of software product lines related to domain engineering, application engineering or management but, feature models are more than only graphical elements to specify software product lines. They contain useful information inside such as the number of potential products being modeled, the core and variant features of the software product line, and so forth. Extracting this type of information manually is an error–prone tedious task and it can be even infeasible to do with large–scale feature models.

The automated analysis of software product lines in general and feature models in particular deals with the computer–aided extraction of information from the software product line that can be of important benefit for software product line analysts, designers, programmers or even managers. To automate the analysis of software product lines a model representing the product line is necessary.

The process to automate the analysis of software product lines is the one depicted in Figure §3.1. First, the model representing the software product line is translated into a logical representation such as propositional logic, constraint programming or description logic so that it is possible to benefit from off–the–shelf solvers that automatically analyse the representation, thus making possible to automatically analyse the software product line itself.

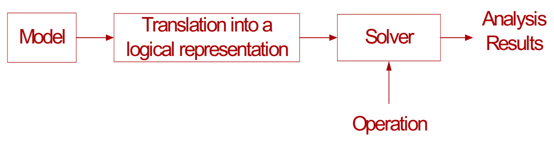


Figure 3.1: *The process for analysing a software product line.*

3.1.1 Scenarios on the automated analysis of feature models

To leverage the automated analysis of feature models it is necessary to have a tool implementing the operations presented in the following sections and the possibility of adding new ones. The operations of analysis performed over feature models will be often composed to provide an useful information. The analysis tool will include all these operations and provide the mechanism of composing them. Let's motivate this with some scenarios.

Imagine a project manager that has to decide about the marketing strategy of a software product line. An automated analysis tool can reveal the number of potential products that is described in the feature model. According to our experience, this data is often surprising to the manager since there are often much more products than expected in the feature model. For instance, in the simple feature model of Figure §2.7 there are already 12 potential products. This number can increase rapidly with the inclusion of more features and relationships. We have dealt with feature models with more than ten thousand potential products which may not be a reasonable value for a marketing strategy manager. However, this information can help him on deciding how many ranges of products the organization will offer. For instance, even if the feature model has a myriad of potential products, the organization could reduce its offer to high end, mid–range and low end products according to features attributes. The automated analysis of the feature model can help on this decision on using an operation for counting the number of products and using an optimization operation. This scenario is depicted in Figure §3.2.

Another possible scenario can be staged in application engineering activities. The aim of application engineering is to build concrete products of a software product line. There are some promising approaches to automate product synthesis from feature models. These approaches are based on *feature oriented programming* (FOP) or model driven development for software product lines (MDD-SPL). The idea is simple: imagine a GUI describing a feature model with check boxes and radio buttons to select the product to be built in a software product line, the aim of FOP or MDD-SPL is that from the selection of features, a product can be automatically synthesized. However, there is an important point here: does the feature model have any errors? this question has to be considered because if the feature model has errors, then this error can be propagated to the final product. There are different type of errors that can be detected in a feature model [91], e.g. detecting if a feature model represent at least one product, detecting if there are some features that are not used in any product, a.k.a. *dead features*. In addition to detect the error, it would be desirable to have an explanation mechanism that explains its source, similarly to error and warning messages in programming languages compilers.

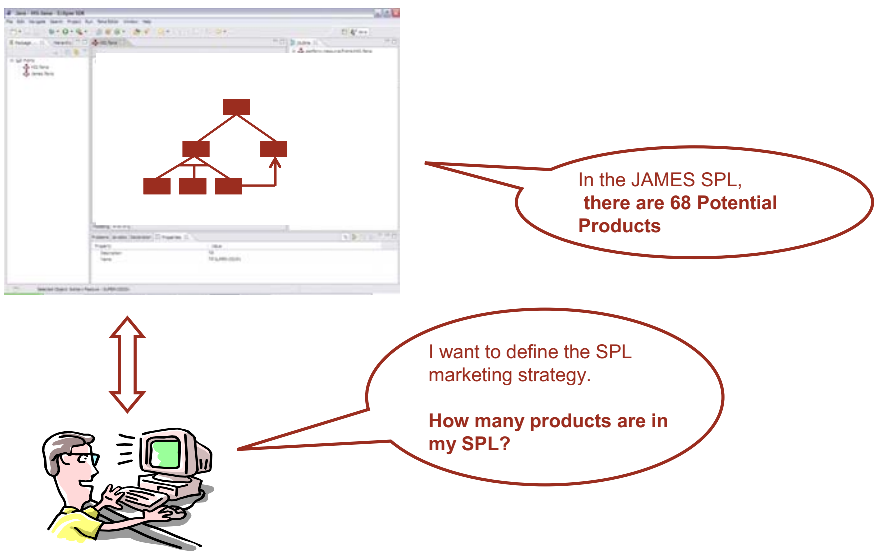

In the JAMES SPL, **there are 68 Potential Products**
I want to define the SPL marketing strategy. **How many products are in my SPL?**

Figure 3.2: *A scenario on the automated analysis of feature models.*

Likewise, when synthesising a product in application engineering there is an activity known as *staged configurations* which consists of configuring a product step by step selecting and deselecting features. It would be of a great help for the application engineer to have an automated tool that when selecting or deselecting features, the tool automatically propagates decisions avoiding incorrect product specifications. For instance, if the feature *automatic* of our car feature model is selected, then *manual* feature has to be automatically deselected by the tool, thus avoiding an incorrect product specification. Moreover, if we are dealing with an extended feature model in which attributes are included, then the propagation could be performed according to criteria on the attributes. For instance, only products with a price lower than a quantity are considered, then some features will be selected and others deselected automatically. Similarly, it is possible to maximize or minimize features attributes according to customers criteria, e.g. looking for the cheapest product or the one with lower time to market. An automated tool able to perform these type of operations has to rely on the automated analysis of feature models as well.

Let's finish with another scenario that also shows the application of the automated analysis. One of domain engineering activities is to define the software product line architecture. If the architecture has to be built according to a given feature model, of course, once again we have to be sure that the feature model has no internal errors, therefore in this scenario again automated analysis can help. However, assuming that the feature model is correct, we can still analyse it to extract useful information. The domain engineer can wonder about the order in which he has to build the elements of the software product line. A first approach is to know the set of core features (those that appear in all products) and start building these features. The set of core features can be extracted automatically from a feature model by an automated tool platform. This operation has been used to define the architecture of software for aerospace missions [78, 79].

In recent years, some contributions have been made to the state of the art within the automated analysis of feature models. They can be grouped in four main sets. First, there are some proposals based on propositional calculus where feature models are translated into propositional formulas. Secondly, description logic is proposed as the formal base and existing description logic reasoners are used to accomplish several automated tasks. Third, we proposed the use of constraint programming where feature models are translated into a constraint satisfaction problem. Finally there are proposals that use ad–hoc solutions defining their own algorithms or using their own tools where the formal base is not clearly stated.

The automated analysis of feature models is still not a mature research field. As a matter of fact, there is not a consensus on the different operations that can be performed.

In this Chapter, we first summarise the different operations identified over feature models until now. We distinguish between operations of analysis and modification. Subsequently, we summarise the four different approaches for the automated analysis of feature models.

3.2 Operations of analysis

We group as operations of analysis those that observe the properties of a feature model without modifying it. This type of operations take a feature model as an input and provide a response.

3.2.1 Determining if a product is valid for a feature model

This operation takes as input a feature model and a product and it returns a value determining if the product belongs to the feature model or not.

For instance, the product P_1, described below, does not belong to the feature model of Figure §2.7 because automatic and manual transmission can not be in a product at the same time. Meanwhile, the product P_2 does belong to the feature model of Figure §2.7.

P_1 = { Car, Body, Transmission, Automatic, Manual, Engine, Electric }

P_2 = { Car, Body, Transmission, Manual, Engine, Electric } .

This operation can be useful for an SPL analyst in order to verify if a product described as a set of features is available in an SPL.

3.2.2 Determining if a feature model is void

This operation takes as input a feature model and returns a value determining whether such feature model is void or not. A feature model is not void if the feature model represents at least one product. A basic feature model without cross–tree constraints can not be void because it is impossible to include contradictions within the model without the use of cross–tree constraints, i.e. a feature model can only be void if cross–tree constraints are included.

The feature models of figure §3.3 are void since they represent no products.

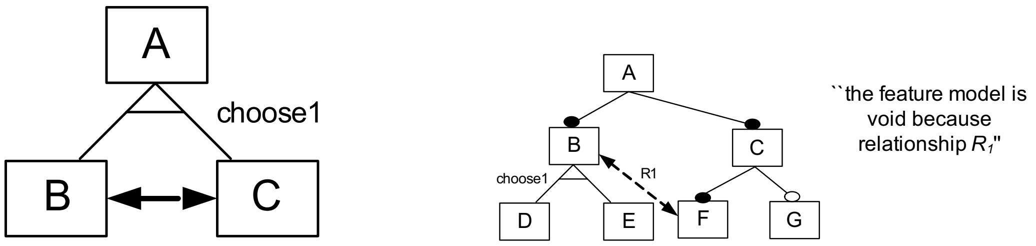


Figure 3.3: *Void feature models.*

This operation is necessary for the automated analysis of feature models specially when coping with large–scale feature models because the debugging of big feature models is recognized to be an error–prone task if it is performed manually [9, 65]. By debugging of feature models we mean the localization and explanation of errors on feature models [91].

3.2.3 Obtaining all the products

This operation takes as input a feature model and returns all valid products of the model. Although this operation is sometimes practical when the number of products is low, it is often unfeasible to perform for feature models with a big number of potential products. This can be used to know if a feature model is void because a feature model is not void iff the set of products retrieved by the *all products* operation is not empty. In other words, a feature model is void iff the all products operations returns an empty set. However, it is important to note that if we just want to know if a feature model is void, you would only need to look for one product, not all of them.

For instance, obtaining all the products in the feature model of Figure §2.7 retrieves this set of products :

{Car, Body, Transmission, Automatic, Engine, Electric},
{Car, Body, Transmission, Manual, Engine, Electric},
{Car, Body, Transmission, Automatic, Engine, Gasoline},
{Car, Body, Transmission, Manual, Engine, Gasoline},
{Car, Body, Transmission, Automatic, Engine, Electric, Gasoline},
{Car, Body, Transmission, Manual, Engine, Electric, Gasoline},
{Car, Body, Transmission, Automatic, Engine, Electric, Cruise},
{Car, Body, Transmission, Manual, Engine, Electric, Cruise},
{Car, Body, Transmission, Automatic, Engine, Gasoline, Cruise},
{Car, Body, Transmission, Manual, Engine, Gasoline, Cruise},
{Car, Body, Transmission, Automatic, Engine, Electric, Gasoline, Cruise},
{Car, Body, Transmission, Manual, Engine, Electric, Gasoline, Cruise}

This operation is not often used when analysing feature models but it is useful to define and perform other analysis operations.

3.2.4 Determining if two feature models are equivalent

This operation takes as input two feature models and returns a value determining if the feature models are equivalent. Two feature models are equivalent if they both represent the same set of products.

For instance, the feature models of Figure §3.4 are equivalents since the set of products specified by them are the same: { { A, B, C} , { A, B, C, D} }.

We distinguish between partial and total equivalence:

- Total equivalence. Two feature models are total equivalent if they represent the same sets of products and the set of features of both of them are the same as well.

- Partial equivalence. Two feature models are partially equivalent if they represent the same sets of products but not necessarily they are expressed using the same set of features. This can be due to the fact that one of the feature models has dead features inside (see Section §3.2.12). For instance, the feature models of Figure §3.5 are partial equivalent but not total equivalent because they both have the same set of products { { A, B, C, D} } but not the same set of features.

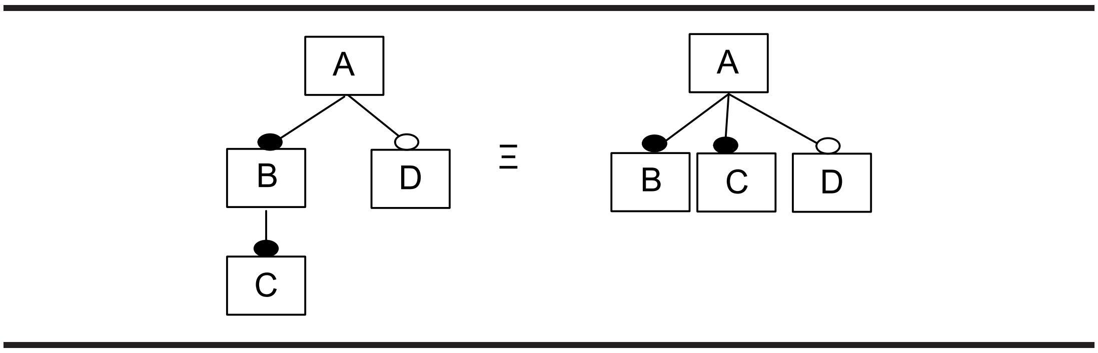


Figure 3.4: *Total equivalent feature models.*

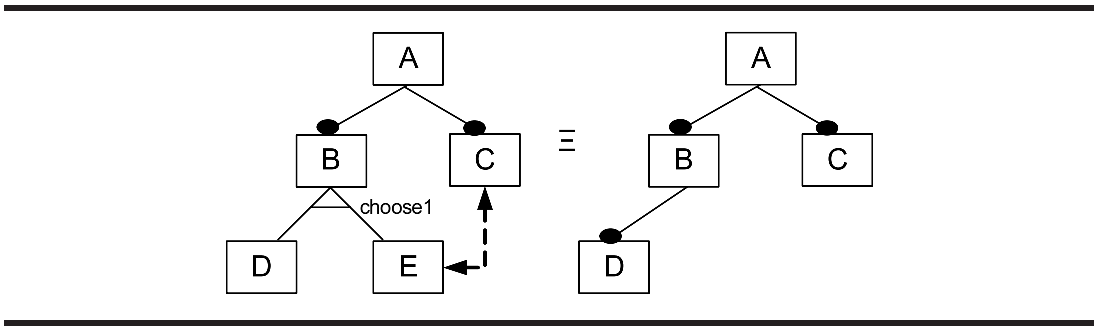


Figure 3.5: *Partial equivalent feature models.*

This operation of analysis can be useful to compare two different feature models or to verify whether changes on a feature model (for example, to simplify it) keep the feature model equivalent.

3.2.5 Retrieving the core features

This operation takes as input a feature model and returns the set of features that appear in all the products. This set of features is known as *core features*. For instance in the feature model of figure §2.7 the set of core features is: { Car, Body, Transmission, Engine } .

This operation can be useful to determine which are the most important features in an SPL and define preferences on building features [79, 92]

3.2.6 Retrieving the variant features

This operation takes as input a feature model and returns the set of features that do not appear in all the products. This set of features is known as *variant features*. For instance in the feature model of figure §2.7 the set of variant features is: { Cruise, Automatic, Manual, Electric, Gasoline } .

3.2.7 Calculating the number of products

This operation takes as input a feature model and returns the number of products of a feature model. This operation can be related to determine whether a feature model is void, because a feature model is not void iff the number of products of the feature model is greater than zero.

This operation reveals information about the flexibility and complexity of the software product line [23, 48, 49, 102]. A big number of potential products can reveal a more flexible as well as more complex product line. For instance, the number of products in the feature model of Figure §2.7 (page 33) is 12.

3.2.8 Calculating variability

This operation takes as input a feature model and returns the ratio between the number of products of the feature model and the number of potential products if all features could be combined with all other without restriction. The number of potential products is defined by $2^n - 1$ being n the number of features that are taken into account. The root is often not taken into account and sometimes only leave features are considered. This operation can help in the analysis of feature models since a big factor would represent a flexible product line meanwhile a low factor would represent a more compact product line.

For instance, the variability of the feature model of Figure §2.7 considering all the features but the root is:

$$\frac{12}{2^8 - 1} \approx 0,0471$$

3.2.9 Filtering a set of products

This operation takes as input a feature model, a set of features F_i to be included and a set of features F_e to be excluded and returns the set of products with the features of F_i and without the features of F_e. In this operation the original feature model is not changed but the set of all possible products is filtered.

For instance, the set of products of the feature model of §2.7 can be filtered with F_i = { Manual } and F_e = { Cruise } . The resulting products are:

$$\{\text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Electric}\},$$
$$\{\text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Gasoline}\},$$
$$\{\text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Electric}, \text{Gasoline}\}$$

This operation is useful when selecting products within a feature model in a staged configuration process [45] where products are configured step by step selecting and deselecting features.

3.2.10 Calculating commonality

This operation takes as input a feature model and a feature within this feature model and returns a value that represents the percentage of valid products where the feature appears.

This operation can help on deciding the order in which feature are going to be developed [79, 92] since highest commonality in a feature may mind highest priority in the order of development. For instance, these are the commonality of some features of the feature model of Figure §2.7 :

$$Commonality(\text{Manual}) = \frac{6}{12} = 0,5$$

$$Commonality(\text{Gasoline}) = \frac{8}{12} = 0,667$$
$$Commonality(\text{Body}) = \frac{12}{12} = 1$$

3.2.11 Optimizing

This operation takes as input a feature model and what is called an *objective function* and returns the best product according to the criterion established by the function [23]. An objective function is a function that from a product is able to decide how good is that product according to that function.

This operation is chiefly useful when dealing with extended feature models where attributes are added to features. For instance, if attributes are added to the feature model of Figure §2.7 (page 33) representing the cost of development of every feature. It would be possible to select the set of products that are cheapest.

3.2.12 Dead features detection

This operation takes a feature model as input and returns a set of *dead features* if any. A non void feature model can have internal inconsistencies in the form of dead features. A dead feature is a feature that never appears in any product of a feature model [48, 91, 93, 109].

Detecting those dead features is yet another challenge to be solved by automated feature models analysis platforms [11, 93] because, although non void feature models can be considered to be error–free feature models, if there are internal dead features in a non void feature model, then, the feature model is said to still have some errors. Therefore, dead features can be viewed as errors in feature models. For instance, all the cases of Figure §3.6 are cases of dead features.

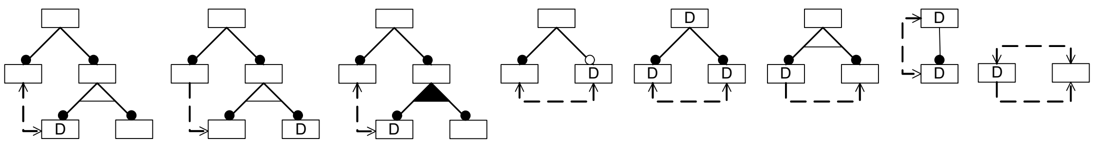


Figure 3.6: *Common cases of dead features.*

3.2.13 Providing explanations

This operation takes as input a feature model and returns an explanation when the feature model is void, a dead feature is detected within the feature model or, in general, an error is detected.

Debugging feature models is an error–prone task if it is performed manually. Usually, the automated tools will be able to detect whether the feature model is void, however, finding the source of the problem that causes the feature model to be void is a key challenge for automated tools [11, 91]. Providing explanations in basic feature models is already a challenging problem and more it is doing so in extended feature models because attributes and relationships among attributes appear.

Unfortunately, providing explanations is not a trivial problem because, rather than telling where the source of the problem is, it is desirable to know the source of the problem as precisely as possible, i.e. the explanation should be minimal. For instance, if a feature model is void, a valid explanation would be "the feature model is the source of the problem". However, the valid information is to pinpoint the relationship that makes the feature model to be void.

For instance, the feature model of figure §3.7 is void. A possible minimal explanation of the source of the problem is: "the feature model is void because relationship R_1". R_1 avoids features B and F appear at the same time meanwhile both features are mandatory and have to appear in all the products which leads to a contradiction.

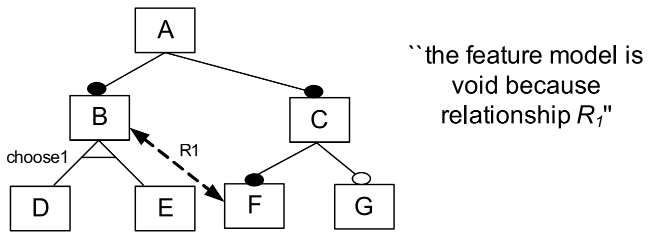


Figure 3.7: *Void feature model with explanations.*

3.2.14 Providing corrective explanations

This operation takes a feature model as input and returns a corrective explanation when the feature model is void or a dead feature is detected. When the user faces a feature model with errors, it is desirable for the automated platform to generate a corrective explanation that, rather than focusing on what is the cause of the problem, explains what can be done to overcome it.

For instance, if the feature model of figure §3.7 is void, a possible corrective explanation would be: "remove relationship R_1" or "change features B or F as *optional* features".

3.3 Operations of modification

All previous operations are based on static feature models, i.e., feature models that do not change over time. However, in practice, it is recognized that feature models evolve over time and these changes have to be contemplated. This type of evolution is defined in the sense that the number of potential products can be reduced in different stages. This is known in the literature as *staged configurations* [44]. During the selection process, a feature model evolve according to selection or deselection of features. The final state of this process would be a feature model that represents only one product or a feature model that represents no products (a void feature model).

We group as operations of modification those that modify the feature model itself. This group of operations take a feature model as an input and return a modified feature model.

3.3.1 Reducing the number of possible products

This operation takes a feature model, a set of features to be selected and a set of features to be deselected as input and modify the original feature model so that the number of possible products of the feature model is reduced by the selection (or deselection) of the features. It is important to note the difference between this operation and the operation of filtering. While filtering does not modify the feature model, this operation does modify it. The way that the feature model is modified will depend on the implementation of the operation. For instance, Batory assigns three states to a feature: selected, deselected and unknown citebatory05-splc.

For instance, in Figure §2.7 (page 33), if automatic transmission is selected the number of possible products is reduced from 12 to 6.

3.3.2 Propagating decisions

This operation takes a feature model as input and returns a modified feature model where some features are automatically selected (or deselected) by the system according to the relationships of the feature model. This operation makes sense during a staged configuration process when the user can select (or deselect) features and the platform should automatically propagate the decision all along the feature model.

For instance, in Figure §2.7, if automatic transmission is selected then manual transmission should be automatically deselected.

3.3.3 Simplification

This operation takes a feature model as input and returns a simplified feature model. One of the drawbacks of feature models is that the same set of products can be specified using different feature models with different relationships among features. This leads to the need of a simplification (so–called normalization) process where any feature model can be translated to a canonical representation [49, 86, 87, 101, 109].

For instance, the feature models of figure §3.8 are two equivalent feature models after a possible simplification process where the simplified feature model can not have at the same level *set* relationships (*alternative* and *or*) and *binary* relationships (*optional* and *mandatory*). The simplification of feature models is not a trivial task and some questions would need to be clarified, for instance, determining if the two models of Figure §3.8 remain equivalent when a new feature is added. However, it is out of the scope of this dissertation to explore all the possible simplification processes of feature models and discuss them.

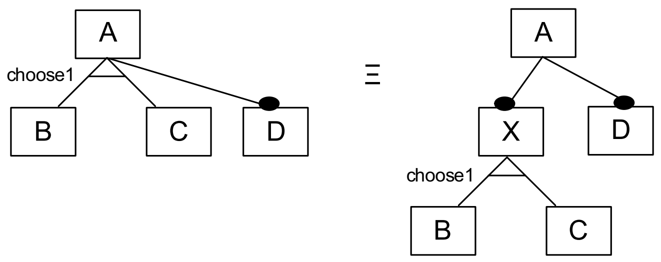


Figure 3.8: *Example of simplified feature model.*

3.4 Automated support for the analysis of feature models

The operations identified in Section §3.2 and §3.3 can be performed using different approaches. Until now, most of these operations have been performed manually, which is not practical especially when dealing with large–scale feature models. Although automated analysis of feature models is still a recent research area and the connection between automated platforms and features models has not been fully appreciated yet [9, 11], there are already some proposals in the literature to automate the analysis of feature models. These proposals can be divided into four different groups. Those based on propositional calculus, those based on description logic, some based on constraint programming and finally others based on ad–hoc solutions.

3.4.1 Propositional logic based analysis

Some authors have proposed the translation of basic feature models into propositional formulas. Recognizing that connection has brought useful benefits since there are many off–the–shelf solvers that automatically analyse propositional formulas, therefore they can be used for the automated analysis of basic feature models. We are not aware of any proposal based on propositional logic to analyse extended feature models where attributes are included in features.

Mannion's proposal

Mannion [73] was the first to connect propositional formulas to feature models. In his work, feature models were used as requirements models for software product lines. Rules for translating such models into propositional formulas were provided and some operations were identified on the automated analysis of feature models. Although a coherent mapping was provided, no tool support was proposed, perhaps, that is why the proposal did not have much impact. Later, in [74] an ad–hoc algorithm was presented to treat those propositional formulas.

Zhang *et al.*'s proposal

Zhang *et al.* [109] suggested the use of an automated tool support based on the SVM System [†1] that allows to analyse propositional formulas. One of the main contributions of this work is the simplification operation (see Section §3.3.3). Moreover, a systematic way to detect dead features (see Section §3.2.12) was provided as well.

Sun *et al.*'s proposal

Sun *et al.* [89] proposed a formalization of feature models using Z [107]. They also proposed the use of the Alloy Analyzer [†2] for the automated analysis of feature models. Specially relevant is the identification and treatment of explanations (see Section §3.2.13) when a feature model is void, i.e. it represents no products.

Batory's proposal

In [9], Batory summarised some of the advances up to date on the automated analysis of feature models. A coherent connection between feature models, grammars and propositional formulas was established. A basic feature model can be represented as a context–free grammar plus propositional formulas for cross–tree constraints what can be the base for the construction of feature model compilers and domain specific languages.

Grammars are presented as a compact representation of propositional formulas, and rules for translating grammars representing feature models into propositional formulas was provided. Furthermore, *Logic Truth Maintenance Systems* (a system that maintains the consequences of a propositional formula) is built for the automated analysis of feature models. Such a system is constructed using a SAT solver and known boolean constraint propagation algorithms.

3.4.2 Description logic based analysis

To the best of our knowledge, there is only one work in the literature that proposes the use of description logic reasoners for the automated analysis of

[†1] www.cs.cmu.edu/~modelcheck/smv.html
[†2] `http://alloy.mit.edu`

feature models [105]. This proposal is based on the translation of feature models into an OWL DL ontology [†3]. OWL DL is a expressive yet decidable sub language of OWL (Ontology Web Language). In that connection, it is possible to use automated tools such as RACER (Renamed ABox and Concept Expression Reasoner[†4]) for the automated analysis of feature models.

3.4.3 Constraint programming based analysis

We were the first to proposing the use of constraint programming for the automated analysis of feature models [20, 23, 24, 27, 93]. A feature model can be translated into a constraint satisfaction problem and using constraint programming techniques it is possible to leverage the automated analyses of feature models. Until now, this is the only proposal that supports the analysis of both cardinality–based and extended feature models and therefore support the optimization operation.

3.4.4 Other proposals

There are some proposals in the literature where the conceptual underpinnings are not clearly exposed. We have decided to group such proposals as ad–hoc solutions. In addition, we have found a proposal that presents a formalization of feature models but does not present any automated support.

Deursen and Klint's proposal

Deursen and Klint [49] proposed a feature description language to describe feature models. From this language, a feature diagram algebra is described based on rules over the ASF+SDF Meta–Environment [68]. Using their system some operations over feature models are automatically performed. After the FODA report in 1990 where feature models were first presented, this is the first paper we have found that explicitly proposes a method for the automated analysis of feature models.

[†3]`www.w3.org/TR/owl-features`

[†4]`www.racer-systems.com`

Cao *et al.*'s proposal

Cao *et al.* [36, 110] presented ad–hoc algorithms for the automated analysis of feature models. Their algorithms are based on the translation of basic feature models into data structures which they claim to be enough to obtain all the products as the base for some other operations. They also present a tool prototype based on their algorithm.

Schobbens *et al.*'s proposal

Schobbens *et al.* [86, 87] survey feature models and study the succinctness, embeddability and naturalness of the different proposals of feature models. However, advanced feature models are not studied. In addition to the survey, the authors propose a generic semantic for feature models to generalize all the works studied. Formal semantics are mentioned using mathematical notations and studies of computational complexity for some operations are presented and proved. Although an automated support is not explicitly presented, it is easy to translate their formal semantics to propositional calculus to analyse feature models.

3.5 Summary

In this chapter, we have presented the operations that we have identified on the analysis of feature models. We have divided them as operations of analysis and operations of modification. Subsequently, we have presented current solutions to automatically analyse feature models. We have divided current proposals on four main groups: propositional logic, description logic, constraint programming and others.

We have published part of these results in some papers, namely: we presented some operations of analysis on feature models in a regular paper in the CAiSE conference [23] that was based on preliminary work that was presented in the International Conference of Software Reuse as a workshop paper [16] and in the 2nd Groningen Workshop on Software Variability Management [22]. Later, we improved and extended [23] and we got a regular paper accepted in the The Seventeenth International Conference on Software Engineering and Knowledge Engineering [24]. In all these papers we proposed the use of constraint programming to analyse feature models. We also published a paper surveying the state of the art on the automated analysis of feature models in the *Jornadas de Ingeniería del Software y Bases de Datos* [25]

Chapter 4

Constraint Programming

Constraint programming represents
one of the closest approaches
computer science has yet made
to the Holy Grail of programming:
the user states the problem,
the computer solves it.[56]

Eugene Freuder,
American scientist

Our goal in this chapter is to provide the reader with a good understanding of constraint programming. In Section §4.1 we provide a brief introduction to constraint programming; in Section §4.2 we rigorously define what a Constraint Satisfaction Problem (CSP) is. In Section §4.3 we describe the main characteristics of CSP solvers and finally, we summarise the chapter in Section §4.4. [†1].

[†1]Some text of this chapter is partially inspired by [8, 32, 35, 59]

4.1 Introduction

Constraint programming has been an important branch of research in artificial intelligence for the last decades [61]. Although constraint programming is based on strong theoretical foundations and it is an active research area in academic contexts, it is also applied in real life problems ranging from scheduling, resource allocation, bioinformatics or even games like *sudoku*. Constraint programming is an interdisciplinary research area since it combines results from different fields such as Artificial Intelligence, Programming Languages or Computational Logic.

The main idea of constraint programming is that the programmer states the problem as constraints between variables and then the computer is responsible for providing the solutions that satisfies the constraints. The way that a solution is retrieved depends on the search strategy so that the same problem can be solved using different strategies.

There are two main branches of constraint programming that are distinguished by how problems are stated.

On the one hand, there are problems that are stated using a finite set of variables, finite domains for those variables and a finite set of constraints over those variables. A problem expressed that way is known as *Constraint Satisfaction Problem* (CSP). A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that all constraints are satisfied simultaneously. For solving a CSP there is a trivial algorithm that generate all possible combinations of values and then it verifies whether each combination satisfies all the constraints or not. If all the constraints are satisfied then the combination is said to be a solution. However this algorithm can take a long time to give the solution in the worst case because its exponential behaviour.

For instance, the following tuple denotes a simple CSP:

$$(\underbrace{\{x, y\}}_{variables}, \underbrace{\{[0..2], [0..2]\}}_{domains}, \underbrace{\{x + y < 4, x - y \geq 1\}}_{constraints})$$

On the other hand, there are some problems that are stated using variables that range over infinite domains like real numbers. It is also common to find more complicated constraints in this type of problems such as non linear equalities. Therefore, for solving these problems another type of algorithms are necessary that use algebraic and numeric methods instead of combinations and search.

In general, when a problem is stated using constraints we may want to find:

i. just one solution, with no preferences. For instance, in previous example a valid solution would be $\{x = 2, y = 1\}$

ii. all solutions. For instance, in previous example:

$$\{\{x = 1, y = 0\}, \{x = 2, y = 0\}, \{x = 2, y = 1\}\}$$

iii. an optimal or good solution. As a matter of fact, we do not often want to find any solution but the best solution. The quality of a solution is determined by the so–called *objective function*. The goal is to find a solution that satisfies all the constraints and minimize or maximize the objective function. Such problems are referred to as *Constraint Satisfaction Optimization Problems* (CSOP), which consist of a standard CSP and an optimization function that maps every solution to a numerical value.

There are many different constraint programming solvers that allow to use well known off–the–shelf algorithms avoiding to rewrite the strategy to search for a solution from scratch.

4.2 Formal definitions of CSPs

As stated earlier, a CSP is defined by a set of variables, a set of domains for those variables and a set of constraints over the variables. To provide a rigorous definition of CSPs we use Z [107] as specification language.

4.2.1 Basic definitions

We define a CSP as a *Z schema* (see Appendix §B). To do so, we have to make some definitions first. We define *VarName* as a *given set* of variable names. Note that determining if a variable name is valid or not is out of the scope of this specification. *Constraint* is also a given set, in this case a given set of constraints. We do not specify the type of constraints that can be used. To do so, a grammar for the language of allowed constraints would be needed and this is again out of the scope of this specification. Finally, *Value* is defined as a given set of values and *Domain* is a finite set of values.

$[VarName]$	[Given set for variable names]
$[Constraint]$	[Given set for constraints]
$[Value]$	[Given set for values to be assigned]
$Domain == \mathbb{F}\ Value$	[A domain is a set of values]

In the Z schema that represents a CSP, we include an assertion ensuring that the variables in the constraint of the CSP are present in the CSP. Therefore, we define a function *variablesIn* that, from a constraint, is able to return the name of the variables that are present in the constraint. This function has to be defined depending on the type of constraints allowed. As said earlier, we do not specify the type of constraints allowed and so we leave the definition open.

Definition 4.1 (variablesIn)

$$variablesIn\_ : Constraint \rightarrow \mathbb{F}\ VarName$$

[concrete definition depends on how constraints are expressed]

With the former definitions we can define now a Z schema for CSPs.

Definition 4.2 (CSP.) *A CSP is defined as a finite set of variables, a function (varDecl) that from a VarName maps the Domain of the variable and a constraint. The constraint will be the conjunction of all the constraints of the CSP. The assertion in the schema is that the variables in the constraints has to be a subset of the variables of the CSP. This way, it is allowed that some variables are not present in any constraint because there may exist variables in a problem that are not constrained.*

CSP

$variables : \mathbb{F}_1\ VarName$

$varDecl : VarName \rightarrowtail\!\!\!\!\!+\ Domain$

$constraint : Constraint$

$dom\ \ varDecl = variables$

$variablesIn\ constraint \subseteq variables$

4.2.2 Solutions of a CSP

A *solution* of a CSP is an *assignment* for every variable of the CSP that satisfies the constraints of the CSP. Let us define what an assignment is as a Z function.

Definition 4.3 (Assignment.) *An assignment is a function that maps every variable of a CSP to a value.*

$$Assignment : VarName \pfun Value$$

To determine if an assignment belongs to the set of solutions of a given CSP, the assignment has to map all the variables to a value on its domain and has to satisfy all the constraints in the CSP. In Z, we define two functions of satisfaction, one at constraint level ($satisfies_c$) and the other at CSP level ($isSolutionOf$).

The first function of satisfaction determines if an assignment satisfies a constraint. This function depends on the type of constraint, therefore we leave its definition open.

Definition 4.4 ($satisfies_c$)

$$\_satisfies_c\_ : Assignment \leftrightarrow Constraint$$

[concrete definition depends on how constraints are expressed]

The other satisfaction function determines if an assignment satisfies a CSP, i.e. it is a solution of the CSP.

Definition 4.5 ($isSolutionOf$.) *For an assignment to be a solution of a CSP, the set of variables of the assignment has to be the same as the set of variables in the CSP. In addition, the value of each variable in the assignment has to be in the domain of the corresponding variable of the CSP and, of course, the assignment has to satisfy the constraint of the CSP.*

$$\_isSolutionOf\_ : Assignment \leftrightarrow CSP$$

$$\begin{aligned}
&\forall a : Assignment;\ csp : CSP \bullet \\
&\quad a\ isSolutionOf\ csp \Leftrightarrow dom\ a = csp.variables\ \wedge \\
&\quad \forall (n, v) : a \bullet\ v \in csp.varDecl(n) \wedge a\ satisfies_c\ csp.constraint
\end{aligned}$$

4.2.3 Operations over CSPs

There are two basic operations defined over CSPs. The first one is to retrieve the set of *solutions* of a CSP and the other is to determine if the CSP is *satisfiable*, i.e. there is at least one assignment that satisfies the constraints of the CSP.

In Z, we define a function returns the set of assignments that conform the set of solutions of a CSP. The set of solutions of a CSP is every valid assignment that satisfies the CSP.

$$\begin{array}{|l}
solutions : CSP \rightarrow \mathbb{F}\,Assignment \\
\hline
\forall\, sol : Assignment;\ csp : CSP \bullet \\
\qquad sol \in solutions\ csp \Leftrightarrow sol\ isSolutionOf\ csp
\end{array}$$

From the former definition we can define now a function that determines whether a CSP is *satisfiable*. There are two ways of defining this function. We can define that a CSP is satisfiable iff the set of solutions of the CSP is not empty.

$$\begin{array}{|l}
satisfiable\_ : \mathbb{P}\,CSP \\
\hline
\forall\, csp : CSP \bullet \\
\qquad satisfiable\ csp \Leftrightarrow solutions\ csp \neq \varnothing
\end{array}$$

Another definition is also possible. In this case, we define a CSP as satisfiable iff there exists at least one assignment that satisfies the CSP. Both definitions are equivalent.

$$\begin{array}{|l}
satisfiable\_ : \mathbb{P}\,CSP \\
\hline
\forall\, csp : CSP \bullet \\
\qquad satisfiable\ csp \Leftrightarrow \exists\, sol : Assignment \bullet\ sol\ isSolutionOf\ csp
\end{array}$$

Another operation that we define over CSPs is the possibility of adding constraints to the CSP. In this case, we define a Z function add_c that has as input a CSP and a set of constraints and returns a new CSP with the constraints added. This function has to be redefined when the type of constraints is defined.

$$\begin{array}{|l}
add_c : CSP \times \mathbb{F}_1\,Constraint \rightarrow CSP \\
\hline
\text{[concrete definition depends on how constraints are expressed]}
\end{array}$$

4.2.4 Constraint optimization problems

The former definitions are valid to find one or all the solutions satisfying a given CSP. That means that all solutions are considered to be equally good. There is another important area of research in constraint programming that deals with the so–called Constraint Optimization Problems (CSOP). What differs a CSOP from a CSP is that while in a CSP we are interested in any solution of the CSP, in a CSOP we do not want just a solution but the best solution. The quality of the solution is determined by an objective function that classifies the goodness of a given solution.

In Z, we define an objective function as a function that takes as input an assignment and returns a real value determining how well the assignment is adapted to that function.

$$Objective : Assignment \rightarrow \mathbb{R}$$

We define also a function called *sortSolutions* that returns a sequence of solutions of a CSP ordered by the criteria of an objective function.

$$\begin{array}{l}
sortSolutions : CSP \times Objective \rightarrow \mathrm{seq}\, Assignment \\
\hline
\forall\, csp : CSP\, ;\ o : Objective;\ s : \mathrm{seq}\, Assignment \bullet \\
\quad sortSolutions(\, csp, o\,) \ = \ s \Leftrightarrow \\
\quad\quad \#s = \#solutions\ csp\ \ \wedge \\
\quad\quad \mathrm{ran}\, s = solutions\ csp\ \ \wedge \\
\quad\quad \forall\, i : 1 \mathrel{..} \#s - 1 \bullet\ Objective\, s(i) \leq Objective\, s(i+1)
\end{array}$$

It is important to notice that these functions are a way of specifying its semantics but not necessarily relate directly to the way that the functions may be implemented in real CSP solvers. For instance, the way a CSP solver retrieve the best solution of a CSP is not often obtained by finding all solutions and then using the objective function to sort them. Instead, CSP solvers will use sophisticated algorithms to ensure that a solution is the best without the need of finding all solutions.

4.3 Constraint Solvers

Once a problem is modeled as a CSP, depending on the nature of the variables, different solvers can be used. There are many solvers both commercial and academic in different languages and platforms. A *CSP solver* determines whether a given CSP is satisfiable or not. If a CSP is satisfiable, the solver can retrieve a solution, all solutions or the best solution depending on the case. Next, different type of solvers depending on the nature of the variables are presented.

4.3.1 SAT solvers

The boolean satisfiability problem (SAT) is a well known problem in computational theory. An instance of the problem is a boolean expression that has three components [41]:

- A set of n boolean variables $x_1 ... x_n$.
- A set of literals that are variables or negation of variables (e.g. x_1 or $\neg x_3$).
- A set of m distinct clauses: $C_1 ... C_m$. Each clause consists of only literals combined by just logical *or* ($\bigvee$) connectives.

The purpose of the satisfiability problem is to determine whether there exists an assignment of truth values (TRUE or FALSE) that makes the following *Conjunctive Normal Form* (CNF) formula satisfiable (every propositional formula can be converted into an equivalent formula that is in CNF [41]):

$$C_1 \bigwedge C_2 \bigwedge \dots C_n$$

A propositional formula is said to be satisfiable if there are some values that can be assigned to its variables in a way that makes the formula true. Determining if there is such assignment was demonstrated to be an NP-complete problem [41].

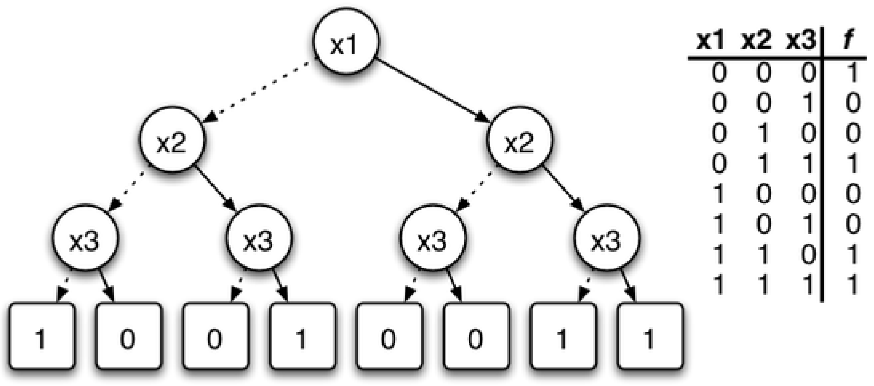
It is important to notice that SAT can be seen as a special case of CSP where each constraint is expressed as a clause and variables are only boolean variables [59].

A *SAT solver* is a software package that basically takes as input a CNF and determines if the formula is satisfiable. Other features are offered by SAT solvers depending on the implementation. There is even an annual competition of SAT solvers where different improvements are presented [†2]

[†2]A list of different SAT solvers and the annual results of the competition can be found at `www.satcompetition.org`

4.3.2 BDD solvers

A Binary Decision Diagram (BDD) is a data structure used to represent boolean functions [34, 35]. A BDD is a rooted, directed, acyclic graph and is composed by a group of decision nodes and two types of terminal nodes called *0–terminal* and *1–terminal*. Each node in the graph represents a variable on a given formula and has two child nodes representing an assignment of the variable to 0 (dashed lines) and 1 (solid lines). All paths from the root to the 1–terminal represents the variable assignments for which the represented boolean function is true meanwhile all paths from the root to the 0–terminal represent the variable assignments for which the represented boolean function is false. Figure §4.1 shows an example.



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 4.1: *Example of a BDD.*

A *BDD solver* is a software package that takes as input a propositional formula and translate it into a BDD. Then, the solver can determine if the formula is satisfiable and can efficiently count the number of possible assignments. Other features are offered by BDD solvers depending on the implementation.

4.3.3 General CSP solvers

The above solvers are restricted to boolean variables, i.e. the problem that is modeled to be the input of BDD or SAT solvers has to be modeled using only boolean variables. In many situations, this approach is not feasible because the gap between the model and the reality can be big. Although in theory any problem that is represented as a CSP can also be represented as a compound of propositional formulas, the language of CSP solvers is often more succinct than the one of BDD or SAT solvers (propositional calculus). This is one of the seeds of the investigation on CSP solvers. It is believed in the constraint programming community (although not proved yet in general) that CSP solvers can be better than SAT or BDD solvers for non-boolean problems.

In general, a CSP solver takes a problem modeled as a CSP as input and determines whether there exists any solution for the problem. Therefore, from a modeling point of view, CSP solvers provide a richer set of variables such as sets, finite integer domains in addition to boolean variables. From the perspective of the operations that a CSP allows, it is important to underline that most of CSP solvers can perform optimization operations according to objective functions. On the contrary, CSP solvers are not good in general for counting the number of solutions [80].[†3]

4.3.4 Search algorithms

When searching for a solution in a CSP, there is a trivial algorithm that generates all the possible combinations of values for the variables and verifies for each one whether it corresponds to a solution of the CSP or not. However, this is a bad algorithm in general. In practice it is much less efficient than algorithms that backtrack on failure, look ahead to detect failure and propagate constraints during search.

There are plenty of advanced and proved algorithms to reduce enormously the search space. In despite of this, developing new algorithms to reduce the search space as much as possible is one of the main areas of research in constraint programming. All these algorithms need to know the order in which variables are considered for assignments in the search algorithm. Once a variable is selected, the order of values, within the variables domain, in which the search algorithm tries to assign values is also important. Variable and value ordering can affect dramatically to the efficiency of constraint satisfaction algorithms.

4.4 Summary

In this chapter, we have presented an introduction to constraint programming and we have also detailed a rigorous definition of constraint satisfaction problems using Z. Finally, we have presented how different constraint solvers can be used depending on the type of constraints of the problem to be solved.

[†3]A list of different CSP solvers can be found at `www.4c.ucc.ie/web/archive/`

Part III
Our Contribution

Chapter 5

Motivation

In don't know of any better motivation than problems

Popular saying

Recently, the automated analysis of feature models has been one of the most interesting research areas in the field of software product lines. However, current solutions are not practical enough since they are not able to cope with advanced feature models, they lack abstract exhaustive definitions and only focus on one type of solver. Our goal in this chapter is to present these problems and motivate the need for a new solution. In Section §5.1 and §5.2 we identify the problems that have to be taken into account in the automated analysis of feature models and in Section §5.3 we analyse current solutions according to the problems identified earlier. In Section §5.4, we put together all the proposals and analyse the capability to solve the problems and argue the need for a new proposal. Finally, we summarise the chapter in Section §5.5.

5.1 Introduction

In recent years, the research community has been paying attention to software product lines, which has led to a huge amount of research papers and industrial experiences. Feature models have been one of the most active areas on software product line research. Notwithstanding these years of research, there is an important weakness that has not been fully solved yet. Feature models have been used mostly as graphical notations to communicate different stakeholders and as a way of expressing commonalities and variabilities in software product lines but something is still missing.

A major challenge in software product line engineering is to automate different processes. One of these processes is *feature model analysis* [11]. In this context, there are several approaches to perform the automated analysis of feature models [9, 23, 36, 49, 73, 89, 93, 105, 109] as presented in Chapter §3. Although some of them address some of the problems that we have identified, none of the solutions seem to be appropriate enough because:

i. All the approaches focus only on feature models.

ii. Formal semantics are not provided.

iii. None of the approaches deal with extended feature models where attributes are introduced.

iv. None of the approaches integrates different solvers.

This argues for a new framework for the automated analysis of software product lines to pave the way for the new generation of feature modelling tools that leverage automated analysis. This is the main motivation of our thesis.

5.2 Problems

Four main problems make it difficult to build tools for the automated analysis of software product lines, namely:

Abstraction. Feature models are only one possible way of modelling software product lines. Current proposals do not open the door to other possible models and only focus on the automated analysis of a certain type of feature models.

Formal semantics. There is not a consensus on the semantics of feature models. However, for an appropriated tool support, a rigorous definition of what a feature model is and which operations are allowed over feature models is needed. Formal semantics are needed in order to rigorously define what software product lines and feature models are. Without semantics, errors and misconceptions can be introduced and tool building is difficult and not rigorous [60].

Support for extended feature models. Automated analyses on basic feature models are covered in some proposals, however a tool should support the automated analysis of advanced feature models where attributes and attribute relationships are considered. The automated analysis of extended feature models is something that is still missing in most of the current works.

Multi Solver. There are some solvers that have been proposed for the implementation of the automated analysis of feature models. Some of them are good for some operations while the performance is poor for some others. A tool that can integrate more than one solver and allows including other solvers would be desirable.

5.3 Analysis of current solutions

Our goal in this Section is to suggest that none of the approaches presented in Chapter §3 addresses the aforementioned problems at the same time.

5.3.1 Abstraction

We have not found any proposal considering the analysis of software product lines at a higher level of abstraction than feature models. We consider that feature models are only one way of modelling software product lines. It is certainly one of the most common ways of doing so. However, the analysis can be abstracted from the specific way of modelling (e.g. feature models) and the operations can be defined at software product line level independently on the specific model to represent variabilities and commonalities. This simplifies the reuse of semantics.

5.3.2 Formal semantics

There are a few works in the literature that provide formal semantics for feature models [86, 87, 89]. However none of them provide those semantics at software product line level or deal with extended feature models. Schobbens *et al.* [86, 87] uses ad–hoc mathematical notations to provide formal semantics for feature models which can lead to misinterpretations. We think that a formal language such as Z [63] is more suitable for providing semantics for feature models than non standard mathematical notations since Z is an ISO standard [63]. As a matter of fact, Sun *et al.* [89], use Z to provide semantics to feature models. However, their semantics are restricted to one type of feature models and are not at a software product line level of abstraction.

5.3.3 Support for extended feature models

The current analysis of feature models is based only on basic feature models. We have proposed to deal with extended feature models as well [11, 23] but in those works we did not include either abstract semantics of software product lines or multi solver support for the analysis of feature models.

5.3.4 Support for basic feature models

Next, we analyse more in depth the current proposals to automatically analyse basic feature models divided in four main groups.

Propositional–based analysis

Propositional–based analyses use different solvers to represent and analyse feature models. Batory [9] proposes the use of SAT solvers. Zhang *et al.* [109] propose the use of SVM system. The SVM system is a system “for checking finite systems against specification in temporal logic”, however the proposal does not use temporal logic but propositional logic. Sun *et al.* [89] propose the use of Alloy Analyzer that internally uses a SAT solver to check model satisfiability. All these proposals are suitable at implementation level and can be considered of a way of performing some operations of analysis over feature models. Nevertheless, we are not aware as a way of using these solvers to analyse advanced feature models and this is the main drawback that we see in these proposals.

Description logic based analysis

Wang *et al.* [105] proposed the use of description logic reasoners to accomplish some of the operations over feature models. We are not aware of any way of using these solvers to obtain all the possible products of a feature model, count the number of products and analyse extended feature models. However, this can be yet another approach to perform some of the operations at implementation level.

Constraint programming based analysis

We have proposed the translation of feature models into CSPs [23]. As a result, we can analyse both basic and extended feature models. We have also proposed the translation of cardinality–based feature models [27] into CSPs. The language used in CSP is more succinct than the one used in SAT or other propositional solvers because CSP solvers allow the use of numerical finite variables such as integers or sets, while for the use of those variables for representing the problem in SAT or BDD approaches, a translation is necessary. Nevertheless, in some operations SAT or BDD solvers can perform better than CSP solvers [26–28], therefore they should be all considered in an automated tool for the analysis of feature models.

5.3.5 Support for multiple solvers

There are some proposals in the literature to deal with the analysis of feature models as presented previously. In these proposals different implementations are presented using different solvers. Some solvers perform better for certain analysis operations while other solvers are better for other operations [26, 28]. Therefore, an automated tool should support the analysis of feature models using different solvers.

5.4 Discussion

A summary of the proposals for the automated analysis of software product lines is depicted in Table §5.1. We have studied five properties. The row Abstraction refers to the level of abstraction considered on the automated analysis of software product lines. A “+” symbol means that the level of abstraction is at SPL level while a ”-“ symbol means that the level of abstraction is at feature models level. The row Formalization refers to formal semantics for feature models. A "+" symbol means that formal semantics are provided while a "-" symbol means that formal semantics are not provided. The row Extended FMs refers to the capability to analyse extended feature models. A "+" symbol means that the proposal deals with extended feature models while a "-" symbol means that the proposal does not deal with extended feature models. The row Basic FMs refers to the capability to analyse basic feature models. A "+" symbol means that the proposal contemplates basic feature models while a "-" symbol means that the proposal does not contemplate basic feature models. Finally the row Multi Solver refers to the implementation of the automated analysis of feature models using different solvers. A "+" symbol means that more than one solver is proposed while a "-" symbol means that none or just one solver was used.

	Batory[9]	Benavides *et al.*[23]	Schobbens *et al.*[86]	Cao *et al.*[36]	Deursen and Klint[49]	Mannion[73]	Sun *et al.*[89]	Wang *et al.*[105]	Zhang *et al.*[109]	FAMA
Abstraction	-	-	-	-	-	-	-	-	-	+
Formalization	-	-	+	-	-	-	-	+	-	+
Extended FMs	-	+	-	-	-	-	-	-	-	+
Basic FMs	+	+	+	+	+	+	+	+	+	+
Multi Solver	-	-	-	-	-	-	-	-	-	+

Table 5.1: *Summary of the proposals for the analysis of software product lines.*

We have studied more in depth the proposals presented in the Table above according to the operations over basic feature models. The summary of the comparison is depicted in Table §5.2. Every row refers to the operations studied in Chapter §3. A "+" symbol means that the operation has been proposed, the "∼" symbol means that although the operation has not been proposed, we envisage that the operation can be performed, finally a "-" symbol means that the operation has not been contemplated and we do not envisage a way to include it in the proposal.

	Batory[9]	Benavides *et al.*[23]	Schobbens *et al.*[86]	Cao *et al.*[36]	Deursen and Klint[49]	Mannion[73]	Sun *et al.*[89]	Wang *et al.*[105]	Zhang *et al.*[109]	FAMA
Valid Product	+	+	+	-	-	+	+	+	∼	+
Void FM	+	+	+	+	+	+	+	+	+	+
All products	+	+	∼	+	+	∼	+	-	∼	+
Equivalent FMs	∼	∼	+	∼	∼	∼	+	∼	∼	+
Core features	∼	∼	∼	-	∼	∼	∼	-	∼	+
Variant features	∼	∼	∼	-	∼	∼	∼	-	∼	+
N. of products	∼	+	∼	+	+	+	∼	-	∼	+
Variability	∼	+	∼	∼	∼	∼	∼	-	∼	+
Filter	+	+	∼	-	∼	∼	∼	∼	∼	+
Commonality	∼	+	∼	-	∼	∼	∼	-	∼	+
Optimization	-	+	-	-	-	-	-	-	-	+
Dead features	-	-	-	-	-	-	-	-	+	+
Explanations	+	-	-	-	-	-	+	+	-	∼
Corrective Explanations	-	-	-	-	-	-	-	-	-	∼
FM reduction	+	-	∼	+	∼	∼	∼	∼	∼	∼
Decision propagation	+	∼	-	-	-	-	∼	-	+	∼
Simplification	∼	-	+	+	+	∼	∼	-	+	∼

Table 5.2: *Summary of the proposals for the analysis of basic feature models.*

From the previous analyses of current solutions we conclude that a new framework able to automate the analysis of software product lines is needed. It should *i*) define the automated analysis at a higher level of abstraction than feature model level, *ii*) provide formal semantics for both software product lines and feature models, *iii*) contemplate extended feature models in addition to basic feature models *iv*) provide support for multiple solvers.

FeAture Model Analyser (FAMA) is our proposal. It provides engineering support for software product line analysts to automate the analysis of feature models. Furthermore, FAMA is extensible in the sense that it can be complemented with *i*) further models other than feature models, *ii*)other operations and *iii*) other solvers that will certainly appear in the future.

5.5 Summary

Our goal in this Chapter was to motivate the reason why we embarked on the development of this thesis. We have analysed the problems involved in the automated analysis of feature models, and have justified that none of the previous proposals found in the literature succeeds in addressing all the problems at a time. This justifies that our contribution is original and advances the state of the art a step forward.

Chapter 6

The FAMA framework

High achievement always takes place
in the framework of high expectation.

Charles F. Kettering, 1876–1958
Engineer

Due to the problems identified in previous chapter, we propose a new framework called FAMA that solves these problems. Our main goal in this chapter is to present the framework and describe the abstract foundation layer. In Section §6.1, we introduce our framework and briefly describe the characteristics of the four layers of FAMA. In Section §6.2, we provide an abstract yet rigorous definition of software product lines. In Section §6.3, we describe the operations of observation at a software product line level. In Section §6.4, we redefine the type feature in order to define attributed features. Finally, in Section §6.5 we summarise the chapter.

6.1 Introduction

As we have stated in the previous chapter, the primary focus of our research work is to provide engineering support so that software product line engineers can automatically analyse software product lines. This is materialized as an abstract framework called *FAMA* that provides a foundation for developing automated feature model analysis tools.

6.1.1 The four–layers framework

We have conceived our framework for the automated analysis of software product lines in four different layers (see Figure §6.1). We have decided to divide the framework in four layers in order to separate different concepts. In the first layer we define software product lines at abstract level and the operations of analysis on software product lines. Then, in the second layer, the first layer is refined by the definition of feature models and keeping the definitions of the operations of the former layer. The second layer serves to define a translation from feature models to CSPs, the operational paradigm that we use to materialize the operations. Finally in the fourth layer, concrete solvers are used to compute the analysis operations defined in the first layer by means of real CSP solvers.

- Abstract foundation layer. It is the basis of the framework. We formally define in this layer what a software product line is and abstractly what are the operations that can be performed in the analysis of software product lines. It is important to underline that in this layer we do not couple to feature models so that the semantics of the operations are independent of how SPLs are modeled (e.g. feature models). This way, if other models are used to describe software product lines, the definitions will remain valid. This layer is described in this chapter. We use Z, a standard specification language [63], to provide formal semantics to SPLs and analysis operations.

- Characteristic model layer. In this layer we define the semantics of software product lines by means of feature models. Most of the definitions of the abstract foundation layer remain the same and only some relations are redefined. If another model is used to describe software product lines, this layer has to be redefined. This layer is described in Chapter §7. We also use Z to provide formal semantics to feature models reusing some of the definitions of the abstract foundation layer.

- Operational paradigm layer. It depends on the modeling technique used in the characteristic model layer (e.g. feature models) and provides a close–to–implementation semantics. In this layer we use CSPs to represent feature models. However, it is important to notice that we use a generic form of CSPs without being coupled to any CSP implementation. This layer is described in Chapter §8. We use Z to define the mapping from feature models to generic CSPs.

- Implementation layer. This layer also depends on the model used in the characteristic model layer and provides a real implementation of the operational paradigm layer, for example JaCoP, SAT4j or JavaBDD solvers are used. This layer is described in Chapter §9.

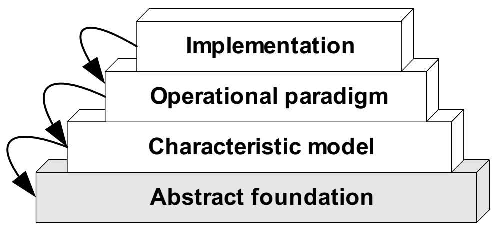


Figure 6.1: *The four layers of the FAMA framework.*

6.2 Abstract foundation layer

A software product line can be defined from several points of view. A quite abstract definition of SPLs is considering it as a non-empty set of features and a *characteristic model* describing allowed *product configurations*. Feature models are widely used as characteristic models, but in this layer we abstract from feature models to define what an SPL is. This allows us to avoid redefining semantics in the case we want to use other characteristic models.

A *product*, considered as a finite non-empty set of features, is a valid product of an SPL if its set of features is a subset of the SPL feature set, i.e. it is configured using only *known features*, and if it is an *instance* of the characteristic model of the SPL, i.e. it is an allowed configuration.

To be presented more formally using Z [107] [†1], we can define two given sets, *Feature* – which can be redefined, for instance to include attributes– and

[†1]see Appendix §B for an overview of the notation

Model –which has to be redefined in the characteristic model layer–, the *Product* type as a finite set of at least one feature and the *isInstanceOf* relation between *Product* and *Model*. This relation does not have a concrete definition because it depends on how features and models are expressed, therefore it has to be redefined in the characteristic model layer.

$[Feature]$ [Given set for features (abstract, to be redefined)]
$[Model]$ [Given set for models (abstract, to be redefined)]
$Product == \mathbb{F}_1\,Feature$ [Definition of Product type]

$\_isInstanceOf\_ : Product \leftrightarrow Model$

[concrete definition depends on how features and models are expressed]

We also need to define a function that returns the set of features involved in the specification of a given characteristic model. This function is useful for specifying that all the features in an SPL must be involved in its characteristic model and vice versa, i.e. that an SPL cannot contain *unbound* features and that a characteristic model must use *all and only* the features in its SPL. This function has to be also redefined in the characteristic model layer.

$featuresIn : Model \rightarrow \mathbb{F}\,Feature$

[concrete definition depends on how features and models are expressed]

Using the previous definitions, an SPL can be defined in Z as the following schema type with a *model* that represent the characteristic model of the SPL and a non empty set of *features*. The assertion that has to hold in the schema is that the set of features in the model has to be the same as the set of features of the SPL.

SPL

$features : \mathbb{F}_1\,Feature$
$model : Model$

$featuresIn\ model = features$

6.3 Operations of observation on abstract SPLs

Using the former definition of SPL we can now formally define operations of observation in the analysis of software product lines. An operation of observation observes the properties of an SPL without modifying it. In this dissertation we only deal with the operations of observation. Notice the relation between this operations and the operations defined in Chapter §3. Meanwhile in Chapter §3 the operations were informally described referred to feature models, in this Section we provide formal semantics to the same operations but at a more abstract level. In this dissertation we only cover the operations pinpointed in Table §5.2 so that we do not cover explanations.

6.3.1 Determining if a product is valid for an SPL

As previously stated, a product is a valid configuration for an SPL if it is configured using the features in the SPL and is an instance of its characteristic model. This can be expressed in Z by means of the following relation:

$$
\begin{array}{|l}
\_isValidFor\_ : Product \leftrightarrow SPL \\
\hline
\forall\, p : Product;\ spl : SPL \bullet \\
\quad p\ isValidFor\ spl \ \Leftrightarrow\ p \subseteq spl.features\ \wedge \\
\qquad\qquad\qquad\qquad\quad p\ instanceOf\ spl.model
\end{array}
$$

6.3.2 Void SPL

An SPL is considered to be void if there is not any product that can be configured using the features of the SPL and its model. This can be expressed in Z by means of the following function.

$$
\begin{array}{|l}
isVoid\_ : \mathbb{P}\, SPL \\
\hline
\forall\, spl : SPL \bullet\ isVoid\ spl\ \Leftrightarrow \nexists p : Product \bullet\ p\ isValidFor\ SPL
\end{array}
$$

Other definition would be possible. In the following definition, an SPL is defined as void if the number of products of the SPL is equal to zero. In this case we use a function *numberOfProducts* that is defined afterwards.

$$
\begin{array}{|l}
isVoid\_ : \mathbb{P}\, SPL \\
\hline
\forall\, spl : SPL \bullet\ isVoid\ spl\ \Leftrightarrow numberOfProducts\ spl = 0
\end{array}
$$

6.3.3 All possible products

Using the *_isValidFor_* relation, the set of all possible products of an SPL, i.e. all valid configurations for an SPL defined using only the SPL features, can be defined as the following function:

$$
\begin{array}{|l}
products : SPL \rightarrow \mathbb{F}\, Product \\
\hline
\forall\, spl : SPL;\; p : Product \bullet \\
\qquad p \in products\ spl \;\Leftrightarrow\; p\ isValidFor\ spl
\end{array}
$$

An alternative definition is also possible that is semantically equivalent but uses another Z syntax to be defined:

$$
\begin{array}{|l}
products : SPL \rightarrow \mathbb{F}\, Product \\
\hline
\forall\, spl : SPL \bullet products\ spl = \{\, p : Product \mid p\ isValidFor\ spl \,\}
\end{array}
$$

6.3.4 SPL total equivalence

Two SPLs are totally equivalent if they represent the same sets of products and the set of features of both of them are the same as well. We use the following relation to express this operation in Z:

$$
\begin{array}{|l}
\_tEquivalentTo\_ : SPL \leftrightarrow SPL \\
\hline
\forall\, spl_1, spl_2 : SPL \bullet \\
\qquad spl_1\ tEquivalentTo\ spl_2 \;\Leftrightarrow\; products\ spl_1 = products\ spl_2 \;\wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad spl_1.features = spl_2.features
\end{array}
$$

6.3.5 SPL partial equivalence

Two SPLs are partially equivalent if they represent the same sets of products but not necessarily they are expressed using the same set of features. This can be due to the fact that one of the SPLs contains dead features. This can be expressed in Z using the following relation:

$$
\begin{array}{|l}
\_pEquivalentTo\_ : SPL \leftrightarrow SPL \\
\hline
\forall\, spl_1, spl_2 : SPL \bullet \\
\qquad spl_1\ pEquivalentTo\ spl_2 \;\Leftrightarrow\; products\ spl_1 = products\ spl_2
\end{array}
$$

It is important to notice that if two feature models are void, they are partially equivalent even though they do not have any feature in common.

6.3.6 Core features

The core features of an SPL are those that appear in all products of the SPL, i.e. features that appear in all valid configurations. This can be expressed in Z by means of the following function:

$$\begin{array}{|l}
core : SPL \rightarrow \mathbb{F}\, Feature \\
\hline
\forall\, spl : SPL \bullet core\ spl = \{ f : spl.feature \mid \forall\, p : products\ spl \bullet f \in p \}
\end{array}$$

Another definition is also possible. In this case, the distributed intersection ($\bigcap$) is used [†2]

$$\begin{array}{|l}
core : SPL \rightarrow \mathbb{F}\, Feature \\
\hline
\forall\, spl : SPL \bullet core\ spl = \bigcap products\ spl
\end{array}$$

6.3.7 Variant features

The variant features of an SPL are those that do not appear in all the products of the SPL, i.e. the features that are not part of the core features. This can be expressed by means of the following function:

$$\begin{array}{|l}
variants : SPL \rightarrow \mathbb{F}\, Feature \\
\hline
\forall\, spl : SPL \bullet variants\ spl = spl.features \setminus core\ spl
\end{array}$$

An alternative definition is also possible:

$$\begin{array}{|l}
variants : SPL \rightarrow \mathbb{F}\, Feature \\
\hline
\forall\, spl : SPL \bullet variants\ spl = \{ f : spl.features \mid \exists\, p : products\ spl \bullet f \notin p \}
\end{array}$$

6.3.8 Number of products

The number of products of an SPL is the number of valid configurations of the SPL. It can be defined in Z using the previously defined *products* function:

$$\begin{array}{|l}
numberOfProducts : SPL \rightarrow \mathbb{N} \\
\hline
\forall\, spl : SPL \bullet numberOfProducts\ spl = \#products\ spl
\end{array}$$

[†2]the distributed intersection of A is the set consisting of all objects belonging to every set in A

6.3.9 Number of potential products of an SPL

The number of potential products of an SPL can be defined as the hypothetical number of products that would be possible to configure if all features could be combined with each other. This definition is useful for specifying other later functions. It can be defined in Z using as the following function:

$$\begin{array}{|l}
potential : SPL \rightarrow \mathbb{N} \\
\hline
\forall\, spl : SPL \bullet potential\ spl = 2^{\#spl.features} - 1
\end{array}$$

6.3.10 Total variability

The total variability of an SPL is defined as the ratio between the number of products of an SPL and the number of potential products of the SPL. Using the former definitions this can be used in Z as the following function:

$$\begin{array}{|l}
tVariability : SPL \rightarrow \mathbb{R} \\
\hline
\forall\, spl : SPL \bullet tVariability\ spl = \dfrac{numberOfProducts\ spl}{potential\ spl}
\end{array}$$

6.3.11 Partial variability

The partial variability of an SPL is defined as the ration between the number of products of an SPL and the possible combinations between the variant features. Using the former definitions this can be used in Z as the following function:

$$\begin{array}{|l}
pVariability : SPL \rightarrow \mathbb{R} \\
\hline
\forall\, spl : SPL \bullet pVariability\ spl = \dfrac{numberOfProducts\ spl}{2^{\#variants\ spl} - 1}
\end{array}$$

6.3.12 Filter

A Filter of an SPL S over a set of features F_i and a set of features F_e is the set of products of S that includes the features on F_i and excludes the features on F_e. This can be defined in Z by means of the following function:

$$filter : SPL \times \mathbb{F}\,Feature \times \mathbb{F}\,Feature \rightarrow \mathbb{F}\,Product$$

$$\begin{array}{l} \forall\, spl : SPL\,;\; f_i, f_e : \mathbb{F}\,Feature;\; p : \mathbb{F}\,Product \bullet \\ \quad filter(\,spl, f_i, f_e\,) \;=\; p \Leftrightarrow \\ \quad\quad p \subseteq products\; spl \wedge \\ \quad\quad \forall\, s : f_i \bullet\; s \in p \wedge \\ \quad\quad \forall\, t : f_e \bullet\; t \notin p \end{array}$$

6.3.13 Commonality factor

The commonality factor of a feature is a number that measures how frequently the feature occurs in the different products of an SPL. It ranges from 0 to 1, e.g. the core features of an SPL have 1 as commonality factor likewise a dead feature has a commonality factor of 0.

$$commonality : SPL \times Feature \rightarrow \mathbb{R}$$

$$\begin{array}{l} \forall\, spl : SPL\,;\; f : Feature \bullet \\ \quad commonality(\,spl, f\,) \;=\; \dfrac{\#\, filter(\,spl, f, \varnothing\,)}{numberOfProducts\; spl} \end{array}$$

6.3.14 Optimization

This operation takes as input a feature model and an objective function and returns the best product according to the criterion established by the function. First, we define what an objective function is and then a function to sort the solutions according to the objective function.

An objective function is a function that takes as input a product and returns a real value determining how well the product is adapted to that function.

$$Objective : Product \rightarrow \mathbb{R}$$

Thus, a sort function is defined as a function that takes as input an SPL and an objective function and returns a sequence of products ordered by the criteria of the objective function.

$$sortProducts : SPL \times Objective \rightarrow \mathrm{seq}\, Product$$

$$\begin{array}{l} \forall\, spl : SPL\,;\; o : Objective;\; p : \mathrm{seq}\, Product \bullet \\ \quad sortProducts(\,spl, o\,) \;=\; p \Leftrightarrow \\ \quad \#p = \#products\; spl \bigwedge \\ \quad \mathrm{ran}\, p = products\; spl \bigwedge \\ \quad \forall\, i : 1 \,..\, \#p - 1 \bullet\; Objective\, p(i) \leq Objective\, p(i+1) \end{array}$$

6.3.15 Dead features detection

Due to the different operators that can be used in feature models, it is possible to find so–called *dead features*. A dead feature is a feature that does not appear in any product of the SPL in despite of being part of the set of features of the SPL. This can be expressed in Z by means of the following function:

$$\begin{array}{|l}
deadFeatures : SPL \rightarrow \mathbb{F}\, Feature \\
\hline
\forall\, spl : SPL \bullet \\
\quad deadFeatures\ spl = \{f : spl.features \mid \forall\, p : products\ spl \bullet\ f \notin p\}
\end{array}$$

6.4 Formal definition of attributed features

An extended feature model is a model that includes more information other than features and features relationships, e.g. the attributes of a feature. We can redefine the type *Feature* in order to include *features attributes*. However, it is important to underline that we redefine it at the abstract foundation layer so that it does not influence in the way that the characteristic model is defined.

To specify attributed features using Z we redefine the type *Feature*. We define a feature as a Z schema. To do so, we define first some given sets. *FeatureName* is the given set of possible names for features. It is out of the scope of this specification to define the regular expression to specify valid names for features. Same way, *AttributeName* is a given set of names of attributes, *Value* is a given set of possible values and *Type* is a finite set of values.

$[FeatureName]$ [Given set for feature names]

$[AttributeName]$ [Given set for attributed names]

$[Value]$ [Given set for values]

$Type == \mathbb{F}\ Value$ [Definition of Type as a set of values]

Using the former definitions we can write an schema to specify features. A feature consists of a *name*; a function *attributes* that maps an *AttributeName* to a *Type*. The attribute name has a value when the feature is selected (*aValue*) and a default value in the case the feature is not selected (*dValue*). We have to assure by means of an assertion that *aValue* and *dValue* are in the domain of the attribute.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Feature}} \\
\hline
name : FeatureName \\
attributes : AttributeName \rightarrow Type \\
aValue : AttributeName \rightarrow Value \\
dValue : AttributeName \rightarrow Value \\
\hline
\mathrm{dom}\, attributes = \mathrm{dom}\, aValue = \mathrm{dom}\, dValue \bigwedge \\
\forall\, a : AttributeName \mid a \in \mathrm{dom}\, attributes \bullet\; aValue\; a \in attributes\; a \bigwedge \\
\forall\, a : AttributeName \mid a \in \mathrm{dom}\, attributes \bullet\; dValue\; a \in attributes\; a \\
\hline
\end{array}
$$

If the definition of feature is changed as above, then the definition of SPL has to be changed as well. We have to include an additional assertion in which it is assured that there are not two features with the same name.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SPL}} \\
\hline
model : Model \\
features : \mathbb{F}_1\, Feature \\
\hline
featuresIn\; model = features \bigwedge \\
\forall\, f_1, f_2 : features\; \bullet \\
\qquad f1 \neq f2 \Rightarrow\; f_1.name \neq f_2.name \\
\hline
\end{array}
$$

Note that this restriction of not allowing to have two features with the same name can be controversial. For instance, it seems to be an error to have the same name for any direct child of a feature (e.g. there can not be two features called A as optional and mandatory features of the same feature P). However, it could be possible, although in most cases confusing, to have two features with the same name that have different parents. However, this depends on the characteristic model layer, so this restriction can be bound also at the characteristic model layer.

6.5 Summary

In this chapter, we have presented the four layers of our framework. In addition, we have provided a rigorous definition of the abstract foundation layer of FAMA in which we have specified some operations of analysis on software product lines using Z at an abstract level without being coupled to any characteristic model. This, as we will see in next chapter, provides a way of reusing semantics.

Chapter 7

Using feature models as characteristic models

The whole of science is nothing more
than a refinement of everyday thinking

Albert Einstein, 1879–1955
Physicists

The characteristic model layer of FAMA is presented in this chapter. We first provide semantics to feature models in Section §7.2 showing some examples in Section §7.3. In Section §7.4, we provide semantics to cross-tree constraints extensions to feature models. In Section §7.5 and Section §7.6, we argue that the semantics for operations and attributed features can be reused in this layer. We finally summarise the chapter and our main contributions in Section §7.7.

7.1 Introduction

In Chapter §6, we described the abstract foundation layer of our framework meanwhile in this Chapter we describe the characteristic model layer. The main difference between these two layers is that in the former we do not specify how the possible configurations of an SPL are defined. In this chapter we use feature models as characteristic models of SPLs. We provide semantics to feature models that allow us to define rigorously how possible configurations of an SPL can be expressed. It is important to remark that there are different feature model notations and semantics as described by Schobbens *et al.* [86]. We have selected to define semantics for basic feature models using the most popular relationships as described in Section §2.2. If other feature model notation is used, then some of the relations defined in this layer would have to be changed.

7.2 Characteristic model layer

Although the characteristic model of an SPL could be defined in different forms, the most widely used is the so–called *feature model* [65]. In a feature model, features are hierarchically organized in *and–or*–like trees, considering non–leaf nodes as compound features. See figure §7.1 for a widely used example that we have already described in Section §2.5.1.

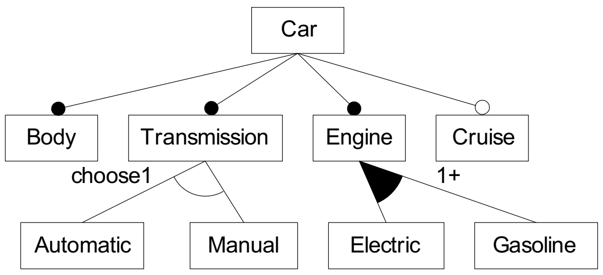


Figure 7.1: *A sample feature model.*

The abstract Z model of an SPL described in the previous Chapter can be enhanced in order to use feature models as the language for expressing SPL characteristic models. In order to do so, we have to redefine three main elements of the abstract foundation layer. The given set *Model* will now define the *abstract syntax* of feature models. The *featuresIn* function has to be defined according to this new syntax. Finally the relation *_isInstanceOf_* will determine if a product is an instance of a characteristic model. These three elements are the ones that has to be redefined if other semantics wants to be used to define models.

We redefine *Model* as a the following Z free type, which represents the abstract syntax of feature models. Notice that *mandatory* and *optional* features are allowed only as children of *and* compound features, thus simplifying semantics.

Definition 7.1

$$\begin{array}{rcl} \mathit{Model} & ::= & \mathit{and}\langle\!\langle \mathit{Feature} \times \mathbb{F}_1\, \mathit{AndChildModel} \rangle\!\rangle \\ & | & \mathit{or}\langle\!\langle \mathit{Feature} \times \mathbb{F}_1\, \mathit{Model} \rangle\!\rangle \\ & | & \mathit{xor}\langle\!\langle \mathit{Feature} \times \mathbb{F}_1\, \mathit{Model} \rangle\!\rangle \\ & | & \mathit{leaf}\langle\!\langle \mathit{Feature} \rangle\!\rangle \end{array}$$

$$\begin{array}{rcl} \mathit{AndChildModel} & ::= & \mathit{mandatory}\langle\!\langle \mathit{Model} \rangle\!\rangle \\ & | & \mathit{optional}\langle\!\langle \mathit{Model} \rangle\!\rangle \end{array}$$

In this abstract syntax, compound features represented as *and*, *or* and *xor* models must have a non-empty set of children models. The auxiliary function *modelOf*, that transforms an *AndChildModel* into a *Model*, is also defined in order to simplify further definitions:

$$\begin{array}{|l} \mathit{modelOf} : \mathit{AndChildModel} \rightarrow \mathit{Model} \\ \hline \forall\, m : \mathit{Model} \bullet \\ \quad \mathit{modelOf}\ \mathit{mandatory}(m) = m\ \wedge \\ \quad \mathit{modelOf}\ \mathit{optional}(m) = m \end{array}$$

Feature models are structurally trees, which implies that a feature cannot appear more than once in the model. In order to make this restriction explicit, the *submodel* function must be defined. This function returns all the models with a given feature as parent.

$$\_submodel\_ : (Model \times Feature) \rightarrow \mathbb{F}\, Model$$

$$\begin{array}{l}
\forall\, m_s : \mathbb{F}\, Model;\ m_{as} : \mathbb{F}\, AndChildModel;\ f_1, f_2 : Feature \bullet \\
\quad f_1 = f_2 \Rightarrow \\
\qquad\quad submodel(\ and(f_1, m_{as}), f_2\ ) \ = \ \{\, and(f_1, m_{as})\, \} \\
\qquad \wedge\ submodel(\ or(f_1, m_s), f_2\ ) \ = \ \{\, or(f_1, m_a)\, \} \\
\qquad \wedge\ submodel(\ xor(f_1, m_s), f_2\ ) \ = \ \{\, xor(f_1, m_a)\, \} \\
\qquad \wedge\ submodel(\ leaf(f_1), f_2\ ) \ = \ \{\, leaf(f_1)\, \}
\end{array}$$

$$\begin{array}{l}
\forall\, m_s : \mathbb{F}\, Model;\ m_{as} : \mathbb{F}\, AndChildModel;\ f_1, f_2 : Feature \bullet \\
\quad f_1 \neq f_2 \Rightarrow \\
\qquad\quad submodel(\ and(f_1, m_{as}), f_2\ ) \ = \ \bigcup\{m_a : m_{as} \bullet submodel(\ modelOf\ m_a, f_2\ )\,\} \\
\qquad \wedge\ submodel(\ or(f_1, m_s), f_2\ ) \ = \ \bigcup\{m : m_s \bullet submodel(\ m, f_2\ )\,\} \\
\qquad \wedge\ submodel(\ xor(f_1, m_s), f_2\ ) \ = \ \bigcup\{m : m_s \bullet submodel(\ m, f_2\ )\,\} \\
\qquad \wedge\ submodel(\ leaf(f_1), f_2\ ) \ = \ \varnothing
\end{array}$$

In a consistent feature model, all sets of models returned by this function should be either empty or *one–element–only* sets so that a feature can only appears once in a feature model. To ensure this in our specification we define the following assertion that has to hold in which we express that for all models the cardinal of the set of submodels for every feature has to be less than or equal to one.

$$\forall\, m : Model;\ f : Feature\ \bullet\ \#submodel(m, f) \leq 1$$

Once we have redefined the *Model* type, the second important part of the specification that has to be defined is the *featuresIn* relation. We distinguish between *compound* and *leaf* features denoting κ–features the set of compound features and λ–features the set of leaf features. The *featureIn* function takes a model and returns the set of features that are in the model. The total set of features is the union of κ–features and λ–features. We may remark that although the definition of this function is a bit complicated, the concept is very simple. We have decided to present the definition of this function in order to make our specification complete.

$$
\begin{array}{|l}
\mathit{featuresIn} : \mathit{Model} \rightarrow \mathbb{F}\,\mathit{Feature} \\
\lambda\text{-}\mathit{featuresIn} : \mathit{Model} \rightarrow \mathbb{F}\,\mathit{Feature} \\
\kappa\text{-}\mathit{featuresIn} : \mathit{Model} \rightarrow \mathbb{F}\,\mathit{Feature} \\
\hline
\forall\, m : \mathit{Model} \bullet \\
\qquad \mathit{featuresIn}\ m \ = \ \lambda\text{-}\mathit{featuresIn}\ m \ \cup\ \kappa\text{-}\mathit{featuresIn}\ m \\
\\
\forall\, f_s : \mathbb{F}\,\mathit{Feature};\ f : \mathit{Feature}; \\
\quad m_s : \mathbb{F}\,\mathit{Model};\ m : \mathit{Model}; \\
\quad m_{as} : \mathbb{F}\,\mathit{AndChildModel};\ m_a : \mathit{AndChildModel};\ \bullet \\
\quad \kappa\text{-}\mathit{featuresIn}\ \mathit{and}(f, m_{as}) \ = \ \{ f \} \cup \\
\qquad\qquad\qquad \bigcup \{ m_a : m_{as} \bullet \kappa\text{-}\mathit{featuresIn}(\mathit{modelOf}\ m_a) \} \\
\quad \land\ \kappa\text{-}\mathit{featuresIn}\ \mathit{or}(f, m_s) \ = \ \{ f \} \cup \bigcup \{ m : m_s \bullet \kappa\text{-}\mathit{featuresIn}\ m \} \\
\quad \land\ \kappa\text{-}\mathit{featuresIn}\ \mathit{xor}(f, m_s) \ = \ \{ f \} \cup \bigcup \{ m : m_s \bullet \kappa\text{-}\mathit{featuresIn}\ m \} \\
\quad \land\ \kappa\text{-}\mathit{featuresIn}\ \mathit{leaf}(f) \ = \ \varnothing \\
\quad \land\ \lambda\text{-}\mathit{featuresIn}\ \mathit{and}(f, m_{as}) \ = \\
\qquad\qquad\qquad \bigcup \{ m_a : m_{as} \bullet \lambda\text{-}\mathit{featuresIn}(\mathit{modelOf}\ m_a) \} \\
\quad \land\ \lambda\text{-}\mathit{featuresIn}\ \mathit{or}(f, m_s) \ = \ \bigcup \{ m : m_s \bullet \lambda\text{-}\mathit{featuresIn}\ m \} \\
\quad \land\ \lambda\text{-}\mathit{featuresIn}\ \mathit{xor}(f, m_s) \ = \ \bigcup \{ m : m_s \bullet \lambda\text{-}\mathit{featuresIn}\ m \} \\
\quad \land\ \lambda\text{-}\mathit{featuresIn}\ \mathit{leaf}(f) \ = \ \{ f \}
\end{array}
$$

Now that we have defined the *featuresIn* function, the final important par of the specification that we have to define is the *isInstanceOf* relation. Notice also the introduction of the auxiliary relation *isInstanceOf*$_a$ for the specification of instances of models of *and* compound features. The *isInstanceOf* relation determines if a product belongs to a model, i.e. a product is an instance of a given feature model. A product is instance of a model if the features of the product are a subset of the features in the model and it is consistent with the relationships of the feature model. For instance, if a feature model has a feature A as mandatory, any product has to have this feature, otherwise, it will not be an instance of the feature model.

Let us specify using Z the *isInstanceOf* relation. In order to do this specification clearer, we have introduced some comments that are explaining subsequently.

$$
\begin{array}{|l}
\_isInstanceOf\_ : Product \leftrightarrow Model \\
\_isInstanceOf_a\_ : Product \leftrightarrow AndChildModel \\
\hline
\forall p : Product;\ f, c : Feature; \\
\quad m_s : \mathbb{F}\, Model;\ m : Model; \\
\quad m_{as} : \mathbb{F}\, AndChildModel;\ m_a : AndChildModel \bullet \\
\quad p\ isInstanceOf\ and(f, m_{as}) \hfill [1] \\
\qquad \Leftrightarrow\ f \in p\ \wedge\ \forall m_a : m_{as} \bullet p\ isInstanceOf_a\ m_a \\
\quad \wedge\ p\ isInstanceOf\ or(f, m_s)\ \Leftrightarrow \hfill [2] \\
\qquad f \in p\ \wedge\ \exists m : m_s \bullet p\ isInstanceOf\ m \\
\qquad\qquad \wedge\ \forall c : roots(m_s) \bullet c \notin p \Rightarrow \neg(p\ isInstanceOf\ m) \\
\qquad\qquad\qquad \wedge\ c \in p \Rightarrow p\ isInstanceOf\ m \\
\quad \wedge\ p\ isInstanceOf\ xor(f, m_s)\ \Leftrightarrow \hfill [3] \\
\qquad f \in p\ \wedge\ \exists_1 m : m_s \bullet p\ isInstanceOf\ m \\
\qquad\qquad \wedge\ \forall c : roots(m_s) \bullet c \notin p \Rightarrow \neg(p\ isInstanceOf\ m) \\
\qquad\qquad\qquad \wedge\ c \in p \Rightarrow p\ isInstanceOf\ m \\
\quad \wedge\ p\ isInstanceOf\ leaf(f)\ \Leftrightarrow\ f \in p \hfill [4] \\
\quad \wedge\ p\ isInstanceOf_a\ mandatory(m)\ \Leftrightarrow\ p\ isInstanceOf\ m \hfill [5] \\
\quad \wedge\ p\ isInstanceOf_a\ optional(m)\ \Leftrightarrow \hfill [6] \\
\qquad\qquad root(m) \notin p \Rightarrow \forall f : featuresIn(m) \bullet f \notin p \\
\qquad\qquad \wedge\ root(m) \in p \Rightarrow p\ isInstanceOf\ m
\end{array}
$$

[1] A model is instance of an *and* model iff the parent of the model (*f*) is in the product (*p*), and *p* is instance of all its children.

[2] A model is instance of an *or* model iff the parent of the model (*f*) is in the product (*p*), and there exists at least one child model (*m*) so that *p* is instance of *m*. Because *p* can be instance of more than one child, we have to ensure that if the root feature of a given child model (*c*) is in the model, then *p* has to be instance of that model. Same way, if the root feature of a given child model is not in the model, then *p* is not instance of the model. The *roots* function is defined afterwards.

[3] A model is instance of an *xor* model iff the parent of the model (*f*) is in the product (*p*), and there exist only one child model (*m*) so that *p* is instance of *m*. Similarly than before, it is also necessary to ensure that if the root feature of a given child model (*c*) is in the model, then *p* has to be instance of that model and if the root feature of a given child model is not in the model, then *p* is not instance of the model.

[4] A product is instance of a *leaf* feature iff the feature is in the product.

[5] A product (p) is instance of a *mandatory AndChildModel* iff p is instance of the model that is inside the *AndChildModel* (see Definition §7.1, page 89).

[6] For a product (p) to be instance of an *optional AndChildModel* either the root of the model is not in p, and therefore, none of the features that are in the model can be in p or the root of the model is in p, and therefore, p has to be instance of the model that is inside the *AndChildModel*.

We have used two auxiliary functions that from a set of Models returns a set with the root of the models to simplify definitions. These functions are specified in Z as follows:

$$\begin{array}{|l}
roots : \mathbb{F}\, Model \rightarrow \mathbb{F}\, Feature \\
\hline
\forall\, ms : \mathbb{F}\, Model;\ f : Feature;\ fs : \mathbb{F}\, Feature \bullet \\
roots(ms) = fs \Leftrightarrow \forall\, m : ms;\ f : root(m) \bullet\ f \in fs
\end{array}$$

$$\begin{array}{|l}
root : Model \rightarrow Feature \\
\hline
\forall\, m : Model;\ ms : \mathbb{F}\, Model;\ f : Feature;\ sacm : \mathbb{F}\, AndChildModel \bullet \\
root\ leaf(f) = f \\
\wedge\ root\ and(f, sacm) = f \\
\wedge\ root\ or(f, ms) = f \\
\wedge\ root\ xor(f, ms) = f
\end{array}$$

Depending on the context a product can be specified using leaf features or also compound features. Notice that in the proposed semantics, products must include not only λ–features but also κ–features. If another semantic is desired the *instanceOf* relation should be redefined. *Model* Z free defines the structure that is allowed when building a feature model meanwhile, *featuresIn* relation defines the semantics of the relationships defined in the *Model* type.

7.3 Some examples

The model in figure §7.1 can be expressed in the feature model abstract syntax of Definition §7.1 as the following:

$$\begin{array}{l}
and(\text{Car}, \{mandatory(leaf(\text{Body})), \\
\qquad mandatory(xor(\text{Transmission}, \{leaf(\text{Automatic}), leaf(\text{Manual})\})), \\
\qquad mandatory(or(\text{Engine}, \{leaf(\text{Electric}), leaf(\text{Gasoline})\})), \\
\qquad optional(leaf(\text{Cruise})) \\
\quad \})
\end{array}$$

Let *m* be feature model in figure §7.1, the following expressions are examples of use of the previous defined relations and functions:

$$\begin{aligned}
&\textit{featuresIn } m \;=\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Manual}, \text{Engine}, \text{Electric}, \text{Gasoline}, \text{Cruise} \,\}\\
&\lambda\text{-}\textit{featuresIn } m \;=\; \{\, \text{Body}, \text{Automatic}, \text{Manual}, \text{Electric}, \text{Gasoline}, \text{Cruise} \,\}\\
&\kappa\text{-}\textit{featuresIn } m \;=\; \{\, \text{Car}, \text{Transmission}, \text{Engine} \,\}\\
&\{\text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Engine}, \text{Electric} \,\}\; \textit{instanceOf } m \quad \text{is true}\\
&\{\text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Manual}, \text{Engine}, \text{Electric} \,\}\; \textit{instanceOf } m \quad \text{is false}\\
&\qquad\qquad\qquad\qquad [\text{because Automatic and Manual can not be at the same time}]\\
&\{\text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Electric}, \text{Gasoline}, \text{Cruise} \,\}\; \textit{instanceOf } m \quad \text{is true}
\end{aligned}$$

$$\begin{aligned}
&\textit{products } m \;=\; \{\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Engine}, \text{Electric} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Electric} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Engine}, \text{Gasoline} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Gasoline} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Engine}, \text{Electric}, \text{Gasoline} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Electric}, \text{Gasoline} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Engine}, \text{Electric}, \text{Cruise} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Electric}, \text{Cruise} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Engine}, \text{Gasoline}, \text{Cruise} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Gasoline}, \text{Cruise} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Automatic}, \text{Engine}, \text{Electric}, \text{Gasoline}, \text{Cruise} \,\},\\
&\quad \{\, \text{Car}, \text{Body}, \text{Transmission}, \text{Manual}, \text{Engine}, \text{Electric}, \text{Gasoline}, \text{Cruise} \,\}\\
&\}\\
&\#\textit{products } m \;=\; 12
\end{aligned}$$

7.4 Cross–tree constraints extensions

Although widely used, feature models as described in previous Sections present a limited expressiveness. Since feature models were first proposed some extensions were introduced. Two of the most usual are the so-called *requires* and *excludes* relationships (see Section §2.2, page 27). In order to include such extensions in this layer model, it is necessary to define and redefine some functions and relations.

First of all, we have to define an *extended model* with cross–tree constraints as the type ($Model_\chi$). A model is extended with cross–tree constraints when it is a feature model plus a set of *extensions*, i.e. *requires* or *excludes* relationships between feature *well–formed–formulas* (WFFs). Feature WFF are logical formulas involving features such us Automatic, Manual $\wedge$ Gasoline or $\neg$ Cruise.

Feature WFFs can be used in *extensions* for expressing conditions like *if a car has automatic transmission and it is electrically powered, it cannot have cruise control*, that would be textually depicted as:

$$\textsf{Automatic} \wedge \textsf{Electric}\ \textit{excludes}\ \textsf{Cruise}$$

and in the proposed abstract syntax as:

$$excludes(and_\Phi(id_\Phi(\textsf{Automatic}), id_\Phi(\textsf{Electric})), id_\Phi(\textsf{Cruise}))$$

The new definitions are the following:

$$Model_X == Model \times \mathbb{F}\, Extension$$

$$\begin{aligned} Extension\ ::=\ & requires\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \\ |\ & excludes\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \end{aligned}$$

$$\begin{aligned} WFF_\Phi\ ::=\ & and_\Phi\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \\ |\ & or_\Phi\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \\ |\ & not_\Phi\langle\!\langle WFF_\Phi \rangle\!\rangle \\ |\ & id_\Phi\langle\!\langle Feature \rangle\!\rangle \end{aligned}$$

7.4.1 Cross–tree constraints semantics

Once cross-tree constraints extensions of feature models are defined, it is necessary to define their semantics. In order to do so, the *instanceOf* relation for extended models (*instanceOf*$_X$) must be defined. Notice that the auxiliary relations *instanceOf*$_\Phi$ and *instanceOf*$_\omega$ define the semantics of the extensions and feature WFFs respectively, which cannot be defined on their own but always in the context of the model they belong to. In this case, the *isInstanceOf*$_X$ relation determines if a product is an instance of a model with cross-tree constraints extensions. A product is an instance of these type of models if it is an instance of the tree–like model and all the extensions.

$$\begin{array}{|l} \_isInstanceOf_X\_ : Product \leftrightarrow Model_X \\ \hline \forall p : Product;\ m : Model;\ x_s : \mathbb{F}\, Extension \bullet \\ \quad p\ isInstanceOf_X\ (m, x_s)\ \Leftrightarrow\ p\ isInstanceOf\ m\ \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall x : x_s \bullet p\ isInstanceOf_\Phi\ (m, x) \end{array}$$

$$\begin{array}{|l}
\_isInstanceOf_{\phi}\_ : Product \leftrightarrow (Model \times Extension) \\
\hline
\forall p : Product;\ m : Model;\ wff_1, wff_2 : WFF_{\phi} \bullet \\
\qquad p\ isInstanceOf_{\phi}\ (m, requires(wff_1, wff_2)\ ) \Leftrightarrow \\
\qquad\quad p\ isInstanceOf_{\omega}\ (m, wff_1) \Rightarrow p\ isInstanceOf_{\omega}\ (m, wff_2) \\
\quad \wedge\ p\ isInstanceOf_{\phi}\ (m, excludes(wff_1, wff_2)\ ) \Leftrightarrow \\
\qquad\quad p\ isInstanceOf_{\omega}\ (m, wff_1) \Rightarrow \neg\ p\ isInstanceOf_{\omega}\ (m, wff_2)
\end{array}$$

$$\begin{array}{|l}
\_isInstanceOf_{\omega}\_ : Product \leftrightarrow (Model \times WFF_{\phi}) \\
\hline
\forall p : Product;\ m : Model;\ wff_1, wff_2 : WFF_{\phi};\ f : Feature \bullet \\
\qquad p\ isInstanceOf_{\omega}\ (m, and_{\phi}(wff_1, wff_2)\ ) \Leftrightarrow \\
\qquad\quad p\ isInstanceOf_{\omega}\ (m, wff_1) \wedge p\ isInstanceOf_{\omega}\ (m, wff_2) \\
\quad \wedge\ p\ isInstanceOf_{\omega}\ (m, or_{\phi}(wff_1, wff_2)\ ) \Leftrightarrow \\
\qquad\quad p\ isInstanceOf_{\omega}\ (m, wff_1) \vee p\ isInstanceOf_{\omega}\ (m, wff_2) \\
\quad \wedge\ p\ isInstanceOf_{\omega}\ (m, not_{\phi}(wff_1)\ ) \Leftrightarrow \\
\qquad\quad \neg\ p\ isInstanceOf_{\omega}\ (m, wff_1) \\
\quad \wedge\ p\ isInstanceOf_{\omega}\ (m, id_{\phi}(f)\ ) \Leftrightarrow \\
\qquad\quad p\ isInstanceOf\ (\mu\, m : submodel(m, f))
\end{array}$$

In the definition of the $\_isInstanceOf_{\omega}$ function, the last expression uses a Z *definite description* over the set returned by the *submodel* function. A definite description, as defined in [107], is an expression that denotes the unique object in a set that satisfies a given predicate (none, i.e. *true* in our case).

7.5 Operations over feature models

All the definitions of this Chapter allow to reuse all the operations defined over software product lines in Chapter §6 so that the formal definitions of those operations remain valid for the characteristic model layer. This allows us to open the door to other semantics of feature models or other type of characteristic models of SPLs. If another type of characteristic model wants to be introduced we have to redefine only the type *Model* as well as the *featuresIn* function and *_isInstanceOf_* relation which simplify the reuse of semantics.

7.6 Feature models with attributes

Another proposed extension of feature models is the inclusion of feature attributes [23]. We already defined attributed features in Section §6.4 (page 84). Therefore, a feature model extended with attributes is a feature model as described in this chapter but taking the definition of attributed feature described in Section §6.4. This again provides a flexible way of reusing semantics because the definition of attributed features does not depend whether feature models are used or not as characteristic models.

It is important to note that we define feature attributes, domains and a way of assigning values, however, we do not couple with attributes relationships. Trying to define a language for defining attributes relationships would be a work enough for another dissertation as the one by Ruiz-Cortés [85].

7.7 Summary

In this chapter, we have provided semantics to basic feature models using Z. Due to the level of abstraction of the abstract foundation layer of FAMA, in this layer, we have showed how it is possible to reuse semantics for operations of analysis on feature models.

We have published part of these results in some papers, namely: we presented some operations of analysis on feature models in our seminal paper in the CAiSE conference [23] that was based on preliminary work in [16] and [22]. Later, we improved and extended [23] and we got a regular paper accepted in [24]. However, we did not provide formal semantics to feature model as we have in this chapter.

Chapter 8

Using constraint programming to analyse feature models

That's the operative thing right now.

Keith Williams, 1958–
British architect

After the abstract foundation and characteristic model layer of FAMA described in previous chapters, in this chapter we describe the operational paradigm layer of. In this layer, we provide a translation from feature models semantics to CSP semantics. However, CSP semantics are not related to any CSP solver implementation. This chapter is structured as follows: In Section §8.1 we provide a brief introduction to the chapter, then in Section §8.2, we describe the main idea of how to translate a feature model into a CSP; in Section §8.3 we extend the definitions of CSPs of Chapter §4 in order to define the constraints we need to represent feature models as CSPs and then we define some functions to transform a feature model into a CSP. In Section §8.4, we define a function to translate feature models with cross–tree constraints into a CSP. Later, in Section §8.5, we redefine some operations of analysis using CSP semantics. In Section §8.6, we demonstrate that determining if a feature model is void is an NP–complete problem. Finally, we summarise the chapter and our main contributions in Section §8.7

8.1 Introduction

In this Chapter we describe the operational paradigm layer of our framework. The definitions on this layer depend on the model defined in the characteristic model layer. In this layer, we provide close–to–implementation semantics of feature models translating a feature model into a CSP. However, it is important to remark that we use a generic form of CSPs without being coupled to any concrete CSP implementation. In Chapter §9 we provide specific translation for specific solvers.

8.2 Feature models as CSPs

Feature models can be represented as CSPs. Recognizing this connection it is possible to use off-the–shelf CSP solvers to automatically analyse feature models. We first informally describe the general mapping between a feature model and a CSP and in next Sections we rigorously define this translation.

The mapping of a feature model into a CSP has the following general form: *i*) the features make up the set of variables, *ii*) the domain of each variable is the same: $\{true, false\}$ (or $\{0, 1\}$ depending on type of the variables allowed by the solver) and *iii*) every relationship of the feature model becomes a constraint among its features. The constraints can be expressed in the following way:

- *Mandatory* relationship: Let P be the parent and C the child in a *mandatory* relationship , then the equivalent constraint is: $P \Leftrightarrow C$
- *Optional* relationship: Let P be the parent and C the child in an *optional* relationship, then the equivalent constraint is: $C \Rightarrow P$
- *Or* relationship: Let P be the parent in an *or* relationship and $\mathsf{F}_i \mid i \in [1 \ldots n]$ the set of children, then the equivalent constraint is: $F_1 \vee F_2 \vee \ldots F_n \Leftrightarrow P$.
- *Alternative* relationship: Let P be the parent of an *alternative* relationship and $\mathsf{F}_i \mid i \in [1 \ldots n]$ the set of children, then the equivalent constraint is: $(F_1 \Leftrightarrow (\neg F_2 \wedge \ldots \wedge \neg F_n \wedge P)) \wedge (F_2 \Leftrightarrow (\neg F_1 \wedge \neg F_3 \ldots \wedge \neg F_n \wedge P)) \wedge$
  $(F_n \Leftrightarrow (\neg F_1 \wedge \ldots \wedge \neg F_{n-1} \wedge P))$

- *Requires* relationship: Let A be the feature that *requires* B, then the equivalent constraint is: $A \Rightarrow B$
- *Excludes* relationship: Let A be the feature that *excludes* B, then the equivalent constraint is: $\neg(A \wedge B)$.

  It is important to notice that in the *requires* and *excludes* relationships it is possible to find *well–formed–formulas* (WFFs) in the right and left sides of the operation and not only features. An expression as the following is allowed (see Section §7.4, page 94):

  $$\text{Automatic} \wedge \text{Electric } \textit{excludes} \text{ Cruise}$$

  In this case, A and B would represent the WFF itself following these rules:

- *and* operator: Let A and B two WFF in an *and* operator of a WFF, then the equivalent constraint is: $A \wedge B$
- *or* operator: Let A and B two WFF in an *or* operator of a WFF, then the equivalent constraint is: $A \vee B$
- *not* operator: Let A a WFF in a *not* operator of a WFF, then the equivalent constraint is: $\neg A$

8.3 Rigorous specification

In this Section we provide a rigorous specification of the translation of a feature model into a CSP. We first revise the general definitions of CSP that we provided in Section §4.2 (page 57) and then we define some functions to transform a feature model into a CSP.

8.3.1 Preliminaries

To formally describe the necessary translation of a feature model into a CSP in Z, we should first specify the semantics of the language used to define CSPs. Specifying all the constraints allowed in common CSP solvers would require a large specification or even infinite, since most of the solvers allow you to define your own constraints and then we should specify the semantics of these new constraints. However, in order to provide a more rigorous description of the translation of a feature model into a CSP, we provide a specification of a CSP language that we use to specify feature models as CSPs. We give semantics for that language which make our specification more rigorous. This is a general abstract language of CSPs for feature models in the sense that it defines generic semantics not being specific for any real solver. These constraints can be implemented in a real solver as we describe in next Chapter.

A general specification of CSPs was presented in Section §4.2. Some aspects remained open in that specification because some functions depend on the type of constraints that can be used. Now, we redefine some of the definitions to completely specify semantics of the language of constraints that we need to express a feature model as a CSP. Concretely, we have to redefine the *VarName* and *Value* types as well as the *Constraint* type; the $satisfies_c$ and *variablesIn* function and the function add_c to add new constraints to a CSP.

First, we define that the possible names of variables of a CSP are equivalent to the possible names for features. We also define that the possible values for variables are *true* and *false*. A value of *true* for a variable (i.e. a feature) means that the corresponding feature is present in the product, on the contrary, a value of *false* for a variable means that the corresponding feature is not in the product. After that, to make the specification more readable, we remember Definition §4.2 (page 58):

$$[VarName] == [FeatureName]$$
$$[Value] == \{true, false\}$$

```
┌─ CSP ──────────────────────────────────────
│ variables : 𝔽₁ VarName
│ varDecl : VarName ⇸ Domain
│ constraint : Constraint
├──────────────
│ dom varDecl = variables
│ variablesIn constraint ⊆ variables
└────────────────────────────────────────────
```

We have also to redefine the type *Constraint*. We define a Z free type, which represents the abstract syntax of the language of constraints that we use to translate a feature model into a CSP.

$$\begin{aligned}
Constraint ::= \; & and\langle\langle Constraint \times Constraint\rangle\rangle \\
\mid \; & or\langle\langle Constraint \times Constraint\rangle\rangle \\
\mid \; & not\langle\langle Constraint\rangle\rangle \\
\mid \; & implies\langle\langle Constraint \times Constraint\rangle\rangle \\
\mid \; & biconditional\langle\langle Constraint \times Constraint\rangle\rangle \\
\mid \; & atMostOne\langle\langle \mathbb{F}_1\ VarName\rangle\rangle \\
\mid \; & atLeastOne\langle\langle \mathbb{F}_1\ VarName\rangle\rangle \\
\mid \; & id\langle\langle VarName\rangle\rangle
\end{aligned}$$

We ought to redefine definition §4.1 (*variablesIn* function, page 58) according to the former definition of *Constraint* in order to define which are the variables that are in a *Constraint*:

$$
\begin{array}{|l}
\mathit{variablesIn}\_ : \mathit{Constraint} \rightarrow \mathbb{F}\ \mathit{VarName} \\
\hline
\forall c_1, c_2 : \mathit{Constraint};\ v : \mathit{VarName};\ vs : \mathbb{F}\ \mathit{VarName} \bullet \\
\quad \mathit{variablesIn}\ \mathit{and}(c_1, c_2) = \mathit{variablesIn}(c_1) \cup \mathit{variablesIn}(c_2) \\
\land\ \ \mathit{variablesIn}\ \mathit{or}(c_1, c_2) = \mathit{variablesIn}(c_1) \cup \mathit{variablesIn}(c_2) \\
\land\ \ \mathit{variablesIn}\ \mathit{not}(c_1) = \mathit{variablesIn}(c_1) \\
\land\ \ \mathit{variablesIn}\ \mathit{implies}(c_1, c_2) = \mathit{variablesIn}(c_1) \cup \mathit{variablesIn}(c_2) \\
\land\ \ \mathit{variablesIn}\ \mathit{biconditional}(c_1, c_2) = \mathit{variablesIn}(c_1) \cup \mathit{variablesIn}(c_2) \\
\land\ \ \mathit{variablesIn}\ \mathit{atLeastOne}(vs) = vs \\
\land\ \ \mathit{variablesIn}\ \mathit{atMostOne}(vs) = vs \\
\land\ \ \mathit{variablesIn}\ \mathit{id}(v) = v
\end{array}
$$

Let us remember Definition §4.3 (page 59) that defined an assignment as a function that maps a variable name to a value:

$$
\begin{array}{|l}
\mathit{Assignment} : \mathit{VarName} \rightarrowtail \mathit{Value}
\end{array}
$$

Now, we have to define the *satisfies*$_c$ function since its definition was empty (see Definition §4.4, page 59) according to the constraints allowed by the Z free type *Constraint*. This function provides semantics to the former abstract syntax of the constraint language in the sense that defines when a constraint satisfies an assignment.

$$
\begin{array}{|l}
\_\mathit{satisfies}_c\_ : \mathit{Assignment} \leftrightarrow \mathit{Constraint} \\
\hline
\forall a : \mathit{Assignment};\ c_1, c_2 : \mathit{Constraint}; \\
vs : \mathbb{F}\ \mathit{VarName};\ v : \mathit{VarName} \bullet \\
\quad a\ \mathit{satisfies}\ (\mathit{and}(c_1, c_2)) \Leftrightarrow a\ \mathit{satisfies}\ c_1 \land a\ \mathit{satisfies}\ c_2 \\
\land\ \ a\ \mathit{satisfies}\ (\mathit{or}(c_1, c_2)) \Leftrightarrow a\ \mathit{satisfies}\ c_1 \lor a\ \mathit{satisfies}\ c_2 \\
\land\ \ a\ \mathit{satisfies}\ (\mathit{not}(c_1)) \Leftrightarrow \neg(a\ \mathit{satisfies}\ c_1) \\
\land\ \ a\ \mathit{satisfies}\ (\mathit{implies}(c_1, c_2)) \Leftrightarrow a\ \mathit{satisfies}\ (\mathit{or}(\mathit{not}(c_1), c_2)) \\
\land\ \ a\ \mathit{satisfies}\ (\mathit{biconditional}(c_1, c_2)) \Leftrightarrow \\
\qquad\qquad\qquad a\ \mathit{satisfies}\ (\mathit{and}(\mathit{implies}(c_1, c_2), \mathit{implies}(c_2, c_1))) \\
\land\ \ a\ \mathit{satisfies}\ (\mathit{atMostOne}(vs)) \Leftrightarrow \exists_1\, v : vs \bullet a(v) = \mathit{true} \\
\land\ \ a\ \mathit{satisfies}\ (\mathit{atLeastOne}(vs)) \Leftrightarrow \exists\, v : vs \bullet a(v) = \mathit{true} \\
\land\ \ a\ \mathit{satisfies}\ (\mathit{id}(v)) \Leftrightarrow a(v) = \mathit{true}
\end{array}
$$

Another important operation that we defined over CSPs is the possibility of adding new constraints to the CSP. We did not define this function in the general definition of CSPs because it depends on how constraints are expressed. Adding a constraint to a CSP will keep the same set of variables, the same domain for those variables and the constraint of the CSP will be the conjunction of the constraints to be added and the constraint of the CSP.

$$
\begin{array}{|l}
add_c : CSP \times \mathbb{F}_1\, Constraint \rightarrow CSP \\
\hline
\forall\, csp, csp_r : CSP;\ cs : \mathbb{F}_1\, Constraint;\ c : Constraint \bullet \\
add_c(csp, cs) = csp_r \Leftrightarrow \\
\quad csp_r.variables = csp.variables \land \\
\quad \mathrm{dom}\, csp_r.varDecl = \mathrm{dom}\, csp.varDecl \land \\
\quad \mathrm{ran}\, csp_r.varDecl = \mathrm{ran}\, csp.varDecl \land \\
\quad csp_r.constraint = conjunction(\{ csp.constraint \} \cup cs)
\end{array}
$$

The conjunction of a set of constraints is a new constraint where each constraint is logically related by the *and* operator.

$$
\begin{array}{|l}
conjunction : \mathbb{F}_1\, Constraint \rightarrow Constraint \\
\hline
\forall\, cs : \mathbb{F}_1\, Constraint;\ c : Constraint \bullet \\
conjunction(\{ c \}) = c \\
\land\ conjunction(\{ c \} \cup cs) = and(c, conjuction(cs))
\end{array}
$$

8.3.2 Characteristic model translation

The general form of the mapping from a feature model into the generic CSP that we have defined in previous Section is the one presented in Figure §8.1.

Following, we describe more rigorously this mapping. The first step in the process of translating a feature model into a CSP is to define the set of variables of the CSP. We define a variable in the CSP for each feature in the feature model. For specifying this using Z we first define a function that from an SPL returns a set of variable names.

$$
\begin{array}{|l}
cspVariables : SPL \rightarrow \mathbb{F}\, VarName \\
\hline
\forall\, spl : SPL;\ vars : \mathbb{F}\, VarName \bullet \\
\quad cspVariables\ spl = vars \Leftrightarrow \\
\quad \forall\, f : spl.features \bullet\ f.name \in vars
\end{array}
$$

Thereafter, we have to define the constraints that represent the relationships of the feature model. In our formal specification we define a function

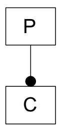
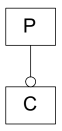
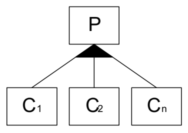
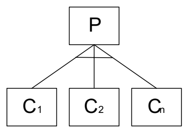
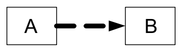
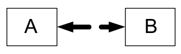
RELATION		Generic CSP
MANDATORY		$biconditional(id(p), id(c))$
OPTIONAL		$implies(id(c),id(p))$
OR		$biconditional(id(p),\ atLeastOne(\{c_1, c_2,..,c_n\}))$
ALTERNATIVE		$biconditional(id(p),\ atMostOne(\{c_1, c_2,..,c_n\}))$
REQUIRES		$implies(id(a), id(b))$
EXCLUDES		$implies(id(a), not(id(b)))$

Figure 8.1: *General mapping from feature models to generic CSP.*

that from a *Model* is able to return a set of constraints. Before doing this, we define different functions for the different elements of a model. For each type of relationship a different constraint is provided.

First, we define a function for *mandatory* relationships that from the parent and child of a *mandatory* relationship returns the corresponding constraint.

$$mConstraint : Feature \times Feature \rightarrow \mathbb{F}\, Constraint$$

$$\forall p, c : Feature \bullet$$
$$mConstraint(p, c) = \{biconditional(id(p.name), id(c.name))\}$$

Second, we define a function for *optional* relationships that from the parent and child of an *optional* relationship returns the corresponding constraint.

$$optConstraint : Feature \times Feature \rightarrow \mathbb{F}\, Constraint$$

$$\forall p, c : Feature \bullet$$
$$optConstraint(p, c) = \{implies(id(c.name), id(p.name))\}$$

Thereafter, we define a function for *or* relationships that from the parent and children of an *or* relationship returns the corresponding constraints. We use an auxiliary function *names* that from a set of features returns the corresponding set of feature names.

$$names : \mathbb{F}\, Feature \rightarrow \mathbb{F}\, FeatureName$$

$$\forall fs : \mathbb{F}\, Feature;\ f : Feature;\ ns : \mathbb{F}\, FeatureName;\ n : FeatureName \bullet$$
$$names(fs) = ns \Leftrightarrow \forall f : fs;\ n : f.name \bullet\ n \in ns$$

$$orConstraint : Feature \times \mathbb{F}_1\, Feature \rightarrow \mathbb{F}\, Constraint$$

$$\forall p : Feature;\ children : \mathbb{F}\, Feature;\ cs : \mathbb{F}\, Constraint \bullet$$
$$orConstraint(p, children) =$$
$$\{biconditional(id(p.name), atLeastOne(names\, children)\}$$

Now, we define a function for *alternative* relationships that from the parent and children of an *alternative* relationship returns the corresponding constraints.

$$xorConstraint : Feature \times \mathbb{F}_1\, Feature \rightarrow \mathbb{F}\, Constraint$$

$$\forall p : Feature;\ children : \mathbb{F}\, Feature;\ cs : \mathbb{F}\, Constraint \bullet$$
$$xorConstraint(p, children) =$$
$$\{biconditional(id(p.name), atMostOne(names\, children)\}$$

The former definitions of functions are the basis of the translation because the corresponding constraint for each relationship is determined. These translations corresponds to the mapping presented in Figure §8.1. Now, we rigorously define the mapping from a feature model to a CSP based on these definitions.

A feature model was defined as a Z free type (see Section §7.1, page 89). It had some relationships and an *AndChildModel* (other Z free type). Therefore, we have to define first a function that from a feature (the parent) and an *AndChildModel* returns a set of Constraints

$$
\begin{array}{l}
acmConstraint : Feature \times AndChildModel \rightarrow \mathbb{F}\, Constraint \\
\hline
\forall f : Feature;\ am : AndChildModel;\ c : Constraint;\ m : Model \bullet \\
acmConstraint(f, mandatory(m)) = \\
\qquad\qquad mConstraint(f, root(m)) \cup cspConstraint(m) \\
\wedge\ acmConstraint(f, optional(m)) = \\
\qquad\qquad optConstraint(f, root(m)) \cup cspConstraint(m)
\end{array}
$$

We can now, define a function that from a *Model* returns a set of constraints corresponding to the relationships of the model.

$$
\begin{array}{l}
cspConstraint : Model \rightarrow \mathbb{F}\, Constraint \\
\hline
\forall f : Feature;\ sacm : \mathbb{F}\, AndChildModel;\ cs : \mathbb{F}\, Constraint;\ sm : \mathbb{F}\, Model \bullet \\
cspConstraint(leaf(f)) = \varnothing \\
\wedge\ cspConstraint(and(f, sacm)) = cs \Leftrightarrow \\
\qquad \forall acm : sacm;\ c : acmConstraint(f, acm) \bullet\ c \subseteq cs\ \wedge \\
\wedge\ cspConstraint(or(f, sm)) = cs \Leftrightarrow \\
\qquad \forall m : sm;\ c : cspConstraint(m);\ cs_{or} : orConstraint(f, roots(sm)) \bullet \\
\qquad\qquad\qquad\qquad c \subseteq cs \wedge cs_{or} \subseteq cs \\
\wedge\ cspConstraint(xor(f, sm)) = cs \Leftrightarrow \\
\qquad \forall m : sm;\ c : cspConstraint(m);\ cs_{xor} : xorConstraint(f, roots(sm)) \bullet \\
\qquad\qquad\qquad\qquad c \subseteq cs \wedge cs_{xor} \subseteq cs
\end{array}
$$

A CSP was defined as a Z schema that had a declaration of a constraint. We assumed that the constraint was the logical conjunction of all the constraints of the CSP. Therefore, the previously defined function is needed to later make the conjunction of the resulting constraints.

We can finally define the mapping of an SPL into a CSP where: the set of variables is the set of features. The domain of the variables is *true* and *false*. The constraint of the model is the conjunction of the model plus and additional constraint that forces the root to be in all the products.

$$
\begin{array}{|l}
cspMapping : SPL \rightarrow CSP \\
\hline
\forall\, spl : SPL;\; csp : CSP;\; v : VarName \bullet \\
cspMapping\ spl = csp \Leftrightarrow \\
csp.constraint = \\
\qquad conjunction(cspConstraint(spl.model) \cup id(root(spl.model).name)) \\
csp.variables = cspVariables\ spl \\
csp.varDecl(v) = \{\, true, false\}
\end{array}
$$

8.3.3 An example of the translation

For example, let us take the model of Figure §2.7 (page 33) that is expressed in the feature model abstract syntax as the following:

$$
\begin{array}{l}
and(\mathsf{Car},\{mandatory(leaf(\mathsf{Body})), \\
\qquad mandatory(xor(\mathsf{Transmision}, \{leaf(\mathsf{Automatic}), leaf(\mathsf{Manual})\})), \\
\qquad mandatory(or(\mathsf{Engine}, \{leaf(\mathsf{Electric}), leaf(\mathsf{Gasoline})\})), \\
\qquad optional(leaf(\mathsf{Cruise})) \\
\qquad \})
\end{array}
$$

The set of constraints (ψ) of the CSP resulting of the previously described algorithm using the abstract syntax of constraints would be as follows:

$$
\begin{array}{l}
\psi = \{id(\mathsf{Car}), \\
\quad biconditional(id(\mathsf{Car}), id(\mathsf{Body})), \\
\quad biconditional(id(\mathsf{Car}), id(\mathsf{Transmission})), \\
\quad biconditional(id(\mathsf{Transmission}), atMostOne(\{\mathsf{Automatic}, \mathsf{Manual}\})), \\
\quad biconditional(id(\mathsf{Car}), id(\mathsf{Engine})), \\
\quad biconditional(id(\mathsf{Engine}), atLeastOne(\{\mathsf{Electric}, \mathsf{Gasoline}\})), \\
\quad implies(id(\mathsf{Cruise}), id(\mathsf{Car})), \\
\quad \}
\end{array}
$$

8.4 Translation of cross–tree constraint extensions

As portrayed in Section §7.4 (page 94), feature model extensions allow to define cross–tree constraints to avoid some feature combinations. The language for extensions was described by means of Z free types as follows:

$$Model_X == Model \times \mathbb{F}\, Extension$$

$$\begin{aligned} Extension &::= requires\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \\ &\mid\ excludes\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \end{aligned}$$

$$\begin{aligned} WFF_\Phi &::= and_\Phi\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \\ &\mid\ or_\Phi\langle\!\langle WFF_\Phi \times WFF_\Phi \rangle\!\rangle \\ &\mid\ not_\Phi\langle\!\langle WFF_\Phi \rangle\!\rangle \\ &\mid\ id_\Phi\langle\!\langle Feature \rangle\!\rangle \end{aligned}$$

We have already defined the translation for a *Model*, now we have to define the translation for *Extension*. We define it by means of the Z functions that from a *well–formed formula* (WFF) returns a set of constraints (*wffConstraint*) and other that from an extension returns the corresponding set of constraints (*extensionConstraint*).

$$\begin{array}{|l} wffConstraint : WFF_\Phi \rightarrow Constraint \\ \hline \forall\, wff_1, wff_2 : WFF_\Phi;\ c : Constraint;\ f : Feature \bullet \\ wffConstraint(and_\Phi(wff_1, wff_2)) = \\ \qquad\qquad and(wffConstraint(wff_1), wffConstraint(wff_2)) \\ \land\ wffConstraint(or_\Phi(wff_1, wff_2)) = \\ \qquad\qquad or(wffConstraint(wff_1), wffConstraint(wff_2)) \\ \land\ wffConstraint(not_\Phi(wff_1)) = not(wffConstraint(wff_1)) \\ \land\ wffConstraint(id_\Phi(f)) = id(f.name) \end{array}$$

$$\begin{array}{|l} extensionConstraint : Extension \rightarrow Constraint \\ \hline \forall\, wff_1, wff_2 : WFF_\Phi;\ e : Extension;\ c : Constraint \bullet \\ extensionConstraint(requires(wff_1, wff_2)) = \\ \qquad\qquad implies(wffConstraint(wff_1), wffConstraint(wff_2)) \\ extensionConstraint(excludes(wff_1, wff_2)) = \\ \qquad\qquad implies(wffConstraint(wff_1), not(wffConstraint(wff_2))) \end{array}$$

Now we can define a function to translate a feature model with cross–tree constraint extensions (*Model*$_x$) into a CSP. We use a new Z function for that:

$$\begin{array}{|l} cspConstraint_X : Model_X \rightarrow \mathbb{F}\, Constraint \\ \hline \forall\, m : Model;\ xs : \mathbb{F}\, Extension;\ cs : \mathbb{F}\, Constraint;\ \bullet \\ cspConstraint_X((m, xs)) = cs \Leftrightarrow cspConstraint(m) \subseteq cs\ \land \\ \qquad\qquad \forall\, x : xs \bullet\ extensionConstraint(x) \in cs \end{array}$$

8.5 Analysis using constraint programming

Once a feature model is translated into a CSP we can use CSP primitives to analyse feature models. Thereafter, we review the operations defined in Chapter §6 and see how they can be solved using CSP primitives. Let ψ_{FM} be the CSP resulting from the translation of a feature model *FM*.

8.5.1 Determining if a product is valid for an FM

A product is a valid configuration of a feature model if its corresponding assignment is a solution of ψ_{FM}. We use an auxiliary function that returns the corresponding *Assignment* of a *Product*. An assignment that represents a products is a function whose domain is the set of variables of the SPL and the values for those variables are mapped to *true* if the feature appears in the product or *false* if the feature does not appear in the product. This is defined by means of the following Z function:

$$\begin{array}{|l}
\mathit{cspAssignment\_} : \mathit{Product} \times \mathit{SPL} \rightarrow \mathit{Assignment} \\
\hline
\forall p : \mathit{Product};\ a : \mathit{Assignment};\ \mathit{spl} : \mathit{SPL};\ f : \mathit{Feature} \bullet \\
\mathit{cspAssignment}\ p = a \Leftrightarrow \\
\qquad \mathrm{dom}\, a = \mathit{spl.features} \land \\
\qquad \forall f : \mathit{Feature} \mid f \in p \bullet a(f.\mathit{name}) = \mathit{true} \land \\
\qquad \forall f : \mathit{Feature} \mid f \in \mathit{spl.features} \setminus p \bullet a(f.\mathit{name}) = \mathit{false}
\end{array}$$

We redefine the semantics of this function by means of the definitions of CSPs. Thus, a product is valid for an SPL using CSPs iff the corresponding assignment of the product is a solution of the CSP.

$$\begin{array}{|l}
\mathit{\_isValidFor\_} : \mathit{Product} \leftrightarrow \mathit{SPL} \\
\hline
\forall p : \mathit{Product};\ \mathit{spl} : \mathit{SPL} \bullet \\
p\ \mathit{isValidFor}\ \mathit{spl} \Leftrightarrow \\
\qquad \mathit{cspAssignment}(p, \mathit{spl})\ \mathit{isSolutionOf}\ (\mathit{cspMapping}\ \mathit{spl})
\end{array}$$

8.5.2 Void feature model

A void feature model is a feature model that does not represent any product, i.e. the number of products of the feature model is equals to zero. Using CSP definitions we can define that a feature model is void if there is not any assignment that is a solution for the corresponding CSP.

$$
\begin{array}{|l}
isVoid\_ : \mathbb{P}\, SPL \\
\hline
\forall\, spl : SPL \bullet isVoid\, spl \Leftrightarrow \\
\qquad \nexists a : Assignment \bullet a\ isSolutionOf\ (cspMapping\ spl)
\end{array}
$$

8.5.3 All possible products

All possible products of an SPL can be defined as all possible solutions of ψ_{FM}. To express this, we have to map the set of all solutions of a CSP into the set of all possible products of an SPL. First, we define a function that from an assignment returns its equivalent product. The equivalent product of an assignment is the set of features where the value of the variable is equals to *true*.

$$
\begin{array}{|l}
toProduct : Assignment \rightarrow Product \\
\hline
\forall\, a : Assignment;\ p : Product;\ f : Feature \bullet \\
toProduct\ a = p \Leftrightarrow \forall f : \operatorname{dom} a \bullet f \in p \Leftrightarrow a(f) = true;
\end{array}
$$

Then, the function to obtain all the possible products of an SPL using CSP definitions is defined in Z as follows.

$$
\begin{array}{|l}
products : SPL \rightarrow \mathbb{F}\, Product \\
\hline
\forall\, spl : SPL;\ ps : \mathbb{F}\, Product;\ a : Assignment \bullet \\
products\ spl = ps \Leftrightarrow \forall\, a : solutions(cspMapping\ SPL) \bullet toProduct\ a \in ps;
\end{array}
$$

8.5.4 Filter

A filter of an SPL S over a set of features F_i and a set of features F_e is the set of products of S that include the features on F_i and excludes the features on F_e. In terms of CSPs that means that the we have to add some constraints to ψ_{FM}. These constraint corresponds to the inclusion of features F_i and the exclusion of features F_e. This can be defined in Z using a function *filter* that from an SPL, a set of features to be included and a set of features to be excluded returns a set of products. This set of products corresponds to the solutions of ψ_{FM} when the corresponding constraints of exclusion and inclusion of features are added to ψ_{FM}. We also use in our Z specification two auxiliary functions $include_f$ and $exclude_f$ to retrieve the constraints of inclusion and exclusion of features.

$$\begin{array}{|l} \mathit{filter} : \mathit{SPL} \times \mathbb{F}\,\mathit{Feature} \times \mathbb{F}\,\mathit{Feature} \rightarrow \mathbb{F}\,\mathit{Product} \\ \hline \forall\, \mathit{spl} : \mathit{SPL}\ ;\ f_i, f_e : \mathbb{F}\,\mathit{Feature};\ \mathit{ps} : \mathbb{F}\,\mathit{Product};\ a : \mathit{Assignment};\ \mathit{csp} : \mathit{CSP} \bullet \\ \mathit{filter}(\ \mathit{spl}, f_i, f_e\ )\ =\ \mathit{ps} \Leftrightarrow \\ \quad \forall\, a : \mathit{solutions}(\mathit{add}_c(\mathit{cspMapping}\ \mathit{SPL}, \mathit{include}_f(f_i) \cup \mathit{exclude}_f(f_e))) \bullet \\ \qquad \mathit{toProduct}\ a \in \mathit{ps} \end{array}$$

$$\begin{array}{|l} \mathit{include}_f : \mathbb{F}\,\mathit{Feature} \rightarrow \mathbb{F}\,\mathit{Constraint} \\ \hline \forall\, \mathit{fs} : \mathbb{F}_1\,\mathit{Feature};\ c : \mathit{Constraint} \bullet \\ \quad \mathit{include}_f(\varnothing) = \varnothing \wedge \\ \quad \mathit{include}_f(f) = \mathit{id}(f.\mathit{name}) \wedge \\ \quad \mathit{include}_f(f \cup \mathit{fs}) = \mathit{id}(f.\mathit{name}) \cup \mathit{include}_f(\mathit{fs}) \end{array}$$

$$\begin{array}{|l} \mathit{exclude}_f : \mathbb{F}\,\mathit{Feature} \rightarrow \mathbb{F}\,\mathit{Constraint} \\ \hline \forall\, \mathit{fs} : \mathbb{F}_1\,\mathit{Feature};\ c : \mathit{Constraint} \bullet \\ \quad \mathit{exclude}_f(\varnothing) = \varnothing \wedge \\ \quad \mathit{exclude}_f(f) = \mathit{not}(\mathit{id}(f.\mathit{name})) \wedge \\ \quad \mathit{exclude}_f(f \cup \mathit{fs}) = \mathit{not}(\mathit{id}(f.\mathit{name})) \cup \mathit{exclude}_f(\mathit{fs}) \end{array}$$

8.5.5 Optimization

We redefine now the sort function of products to perform this function using CSP operations. A sort function was defined as a function that takes as input an SPL and an objective function and returns a sequence of products ordered by the criteria of the objective function.

$$\begin{array}{|l} \mathit{sortProducts} : \mathit{SPL} \times \mathit{Objective} \rightarrow \mathrm{seq}\,\mathit{Product} \\ \hline \forall\, \mathit{spl} : \mathit{SPL}\ ;\ o : \mathit{Objective};\ \bullet \\ \mathit{sortProducts}(\ \mathit{spl}, o\ )\ =\ \mathit{toProducts}(\mathit{sortSolutions}(\mathit{cspMapping}(\mathit{spl}), o)) \end{array}$$

We use an auxiliary function *toProducts* that from a sequence of assignments (the solutions of the *sortSolutions* function of CSPs) returns a sequence of products.

$$\begin{array}{|l} \mathit{toProducts} : \mathrm{seq}\,\mathit{Assignment} \rightarrow \mathrm{seq}\,\mathit{Product} \\ \hline \forall\, a : \mathrm{seq}\,\mathit{Assignment}\ ;\ p : \mathrm{seq}\,\mathit{Product}; \\ \mathit{toProducts}(\ a\ )\ =\ p \Leftrightarrow \\ \quad \#p = \#a \wedge \\ \quad \mathrm{ran}\,p = \mathrm{ran}\,a \wedge \\ \quad \forall\, i : 1\,..\,\#a \bullet\ p(i) = \mathit{toProduct}(a(i)) \end{array}$$

8.5.6 Operations that do not need redefinition

The definition of the following functions remain the same because for their definition one or more of the formed definitions that we redefine in this Chapter were used:

- SPL total equivalence (see Section §6.3.4).
- SPL partial equivalence (see Section §6.3.5).
- Core features (see Section §6.3.6).
- Variants features (see Section §6.3.7).
- Number of products (see Section §6.3.8).
- Total Variability (see Section §6.3.10).
- Partial Variability (see Section §6.3.11).
- Commonality Factor (see Section §6.3.13).
- Dead Features (see Section §6.3.15).

It is important to note that even if the former operations can be performed using previous defined operations using CSP primitives, some of this operations would be of a high order of complexity if CSP solvers are used to perform them. However, there can be ad–hoc algorithms to perform some of the operations that scale better.

8.6 Computational complexity

In this Section we show that, in general, determining if a feature model is void is an NP-complete problem. Since we are using constraint satisfaction problems to represent feature models, it is natural to show this using CSPs.

We first show that any binary CSP can be represented as a feature model. Suppose the CSP P has n variables, $x_1, x_2, ..., x_n$. For the sake of simplicity, we assume that all domains are the same: $\{v_1, v_2, ...v_m\}$. We represent this by a feature model with a root feature P representing the problem P. It has n children, $\mathsf{X_1}$, $\mathsf{X_2}$, ..., $\mathsf{X_n}$, corresponding to the n variables; each of these features is *mandatory* (we must assign a value to every variable).

The feature X_i has *m* children corresponding to the *m* values in its domain, which we label $V_{i1}, V_{i2}, ..., V_{im}$, so that each feature in the model has a distinct name. Exactly one of these features must be selected, so we use an alternative relationship. The 'value' features are the leaves of the feature model.

The constraints of the CSP forbid pairs of variable-value assignments: if the pair of assignments $x_i = v_k$ and $x_j = v_l$ is forbidden by the binary constraint between x_i and x_j, we add an exclusion dependency between the corresponding features V_{ik} and V_{jl}.

Since binary CSPs are in general NP-complete [72], feature models must also be NP-complete. However, note that the feature model constructed to represent a binary CSP is unlike those that arise in practice, and in particular has far more dependencies than we would expect.

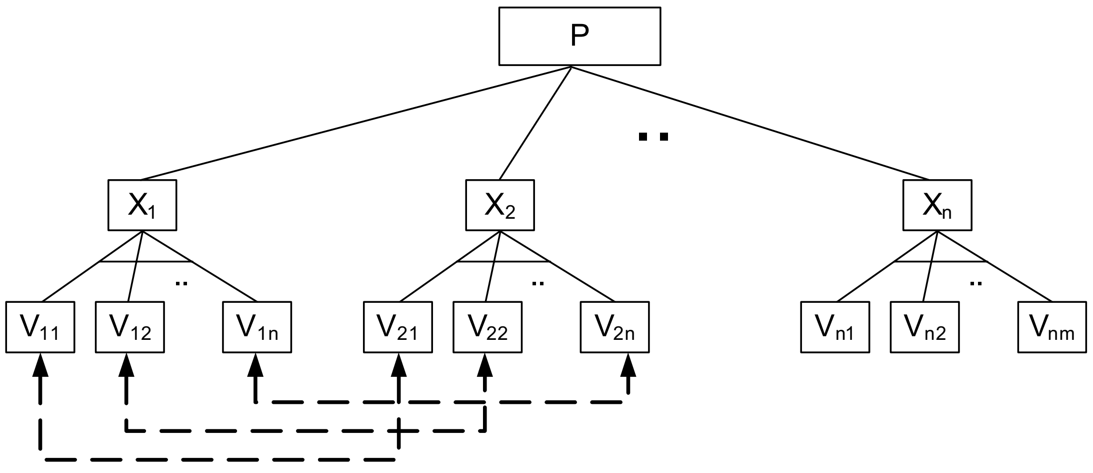


Figure 8.2: *Feature model of a CSP with a constraint* $x_1 \neq x_2$.

8.7 Summary

In this chapter we have presented the operational paradigm layer of FAMA. We have showed how it is possible to translate a feature model into a CSP. We have first presented a rigorous definition of CSPs using Z, later we have defined the translation of both feature models and its extension in terms of previously defined CSPs. We have redefined some operations of analysis using the semantics of CSPs and finally demonstrated some theoretical consequences about complexity.

We have published part of these results in some papers, namely: we presented some operations of analysis on feature models in a regular paper in the CAiSE conference [23] that was based on preliminary work [16, 22]. Later, we improved and extended [23] and we got a regular paper accepted in The Seventeenth International Conference on Software Engineering and Knowledge Engineering [24]. In all these papers we proposed the use of constraint programming to analyse feature models.

Chapter 9

Implementing our framework

An idealist is someone who implements ideas

David Benavides, 1976–

The implementation layer of FAMA is presented in this chapter. We provide the translation of feature models into concrete CSP solvers. First, the translation into general CSP solvers is described in Section §9.2; In Section §9.3 we provide a translation from feature models into CNF, the basic input of a SAT solver. In Section §9.4, the rules for translating a feature model into a BDD structure is presented. Later, in Section §9.5 some performance test are described comparing BDD, SAT and general CSP solvers. We finally summarise the chapter and our main contributions in Section §9.6.

9.1 Introduction

In this Chapter, we describe the implementation layer of our framework. In this layer, we provide a real implementation of the operational paradigm described in Chapter §8. The main difference is that in the operational paradigm layer we translated a feature model into an abstract CSP, i.e. a CSP that was not coupled to any real implementation. The intention of this Chapter is to provide a translation from the abstract CSP to real CSP solvers. As a proof of concepts we use three different type of CSP solvers: general, SAT and BDD solvers. JaCoP and Choco are the general CSP solvers meanwhile SAT4j is used as SAT solver and JavaBDD as BDD solver.

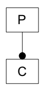
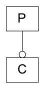
9.2 Translation into general CSP solvers

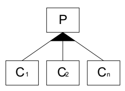
There are several commercial general CSP solvers. One of the major commercial vendors is ILOG that has two versions of CSP Solvers in C++ and Java [†1]. Because it is a commercial solution, we declined to use ILOG solvers licenses in our proof of concepts implementation. In contrast, we selected JaCoP solver [70] because it offers a free license for academic purposes and it was offered by members of the Cork Constraint Computation Centre in order to test our approach. Very similar to this solver is the Choco Constraint System [†2]. Choco is an open source CSP solver written in Java. We also translated a feature model into a problem to this solver. We selected this solver because it seems to be one of the most popular within the research community. Both solvers have similar characteristic in terms of the variables and constraints allowed, therefore the implementation of our mapping was done in a straightforward manner. For JaCoP we used *FDV* variables (FDV stands for Finite Domain Variables) for representing feature variables, while *IntVar* variables were used in the Choco implementation for this purpose.

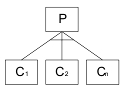
The rules for translating feature models into the JaCoP solver are listed in Figure §9.1. The final representation of the feature model is the conjunction of the translated relationships following the rules of Figure §9.1 plus an additional constraint selecting the root which is included in all products. The column "Generic CSP" corresponds to the abstract language we specified in Chapter §8. For the sake of simplicity we do not present the mapping to the Choco solvers since it is very similar.

[†1]`www.ilog.com`

[†2]`http://choco-solver.net`

RELATION		Generic CSP	JaCoP
MANDATORY		$biconditional(id(p), id(c))$	`XeqY(P,C)`
OPTIONAL		*implies(id(c),id(p))*	`IfThen(XeqC(P,0),XeqC(C,0))`

OR		$biconditional(id(p), atLeastOne(\{c_1, c_2,..,c_n\}))$	`IfThenElse (XgtC(P,0),`
			 `in (Sum({C1,C2,..,Cn}),{1..n}),`
			 `XeqC ({C1,C2,..,Cn},0)`

ALTERNATIVE		$biconditional(id(p), atMostOne(\{c_1, c_2,..,c_n\}))$	`IfThenElse (XgtC(P,0),`
			 `in (Sum({C1,C2,..,Cn}),{1..1}),`
			 `XeqC ({C1,C2,..,Cn},0)`

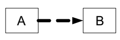
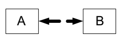
REQUIRES		$implies(id(a), id(b))$	`IfThen(XgtC(A,0),XgtC(B,0))`
EXCLUDES		$implies(id(a), not(id(b)))$	`IfThen(XgtC(A,0),XeqC(B,0))`

Figure 9.1: *Mapping from feature models to JaCoP solver.*

JaCoP works with finite domain variables so we decided to model every features as a FDV variable with a domain $\{0, 1\}$. A solution of the CSP would be a product configuration. A value of 1 in the solution of the CSP represents that the feature appears in the product, and a 0 value means that the features does not appear in that product configuration. The constraints used in JaCoP are:

- `XeqY(a,b)`:is an equality constraint. It constraints the value of variable `a` to be equal to the values of variable `b`.
- `XgtY(a,b)`: constraints the value of variable `a` to be greater than the values of variable `b`.
- `IfThen(c,t)`: if the constraint `c` is satisfied, then the constraint `t` has to be satisfied too.
- `IfThenElse(c,t,e)`: if the constraint `c` is satisfied, then the constraint `t` has to be satisfied too, otherwise, the constraint `e` has to be satisfied.
- `in(e,d)`: The expression `e` has to be in the domain `d`.

9.3 Translation into SAT solvers

As presented in Section §4.3.1 (pag. 62), a propositional formula is an expression consisting of a set of boolean variables (literals) connected by logic operators ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$). The propositional satisfiability problem (SAT) consists of deciding whether a given propositional formula is satisfiable, that is, if logical values can be assigned to its variables in a way that makes the formula true.

A SAT solver allows to analyse propositional formulas. We used SAT4j [†3], an open source SAT solver written in Java. Most of SAT solvers uses a standard CNF (Conjunctive Normal Form) file as input. In our proof of concepts, we translated a feature model into a CNF and wrote a file to serve as input to the solver.

The rules for translating feature models into CNF formulas are listed in Figure §9.2. For the sake of simplicity the CNF file is not presented but the propositional formulas in CNF form to be generated. The final representation of the feature model is the conjunction of the translated relationships following the rules of Figure §9.2 plus an additional constraint selecting the root which is included in all products.

[†3] `www.sat4j.org`

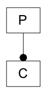
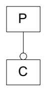
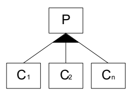
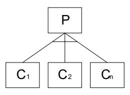
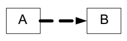
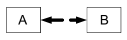
RELATION		SAT
MANDATORY		$(\neg P \vee C) \wedge (\neg C \vee P)$
OPTIONAL		$\neg C \vee P$
OR		$(\neg P \vee C_1 \vee C_2 \vee ... \vee C_n) \wedge (\neg C_1 \vee P) \wedge$ $(\neg C_2 \vee P) \wedge ... \wedge (\neg C_n \vee P)$
ALTERNATIVE		$(C_1 \vee C_2 \vee ... \vee C_n \vee \neg P) \wedge$ $(\neg C_1 \vee \neg C_2) \wedge ... \wedge (\neg C_1 \vee \neg C_n) \wedge (\neg C_1 \vee P) \wedge$ $(\neg C_2 \vee \neg C_3) \wedge ... \wedge (\neg C_2 \vee \neg C_n) \wedge (\neg C_2 \vee P) \wedge$ $(\neg C_{n-1} \vee \neg C_n) \wedge (\neg C_{n-1} \vee P) \wedge (\neg C_n \vee P)$
REQUIRES		$\neg A \vee B$
EXCLUDES		$\neg A \vee \neg B$

Figure 9.2: *Mapping from feature models to SAT solvers.*

9.4 Translation into BDD solvers

As presented in Section §4.3.2 (pag. 63), a BDD is a data structure used to represent boolean functions. It is a rooted, directed, acyclic graph composed by a group of decision nodes and two terminal nodes called 0–terminal and 1–terminal. Each node in the graph represents a variable in a boolean function and has two child nodes representing an assignment of the variable to 0 and 1. All paths from the root to the 1–terminal represents the variable assignments for which the represented boolean function is *true* meanwhile all paths to the 0–terminal represents the variable assignments for which the represented boolean function is *false*.

A BDD solver allows to analyse boolean functions. We used JavaBDD [†4], an open source BDD solver written in Java. It uses a type of variables called BDD to represent nodes in the BDD and it is possible to relate these variables with most used logical connectors (and, or, implies and so on). In our proof of concepts, we translated a feature model into a boolean function and used JavaBDD solver to analyse the corresponding feature model.

The rules for translating feature models into a BDD structure are listed in Figure §9.3, §9.4 and §9.5. In each table the propositional formula and the corresponding BDD is represented. It is important to remark some aspects of the figures. First, the final propositional formula of a given feature model is the conjunction of the translated relationships following the rules of Figure §9.3, §9.4 and §9.5 plus an additional constraint selecting the root which is included in all products. This will lead to have a BDD structure that is not trivial to construct from the partial BDD structures. What we have done is to build the whole propositional formula and then transform it into a BDD structure using JavaBDD package. On the other hand, the BDD structure resulting from translating the *mandatory*, *or* and *alternative* relationships has more nodes than features in the corresponding figure.

[†4]`http://javabdd.sourceforge.net`

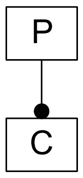
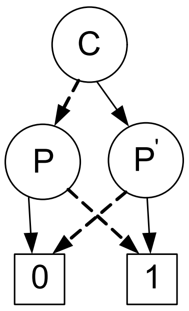
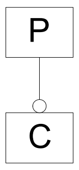
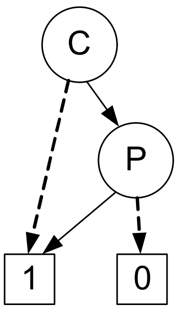
RELATION		Formula	BDD
MANDATORY		$P \leftrightarrow C$	
OPTIONAL		$C \rightarrow P$	

Figure 9.3: *Mapping* mandatory *and* optional *relationships to BDD.*

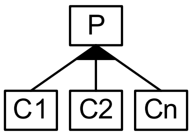
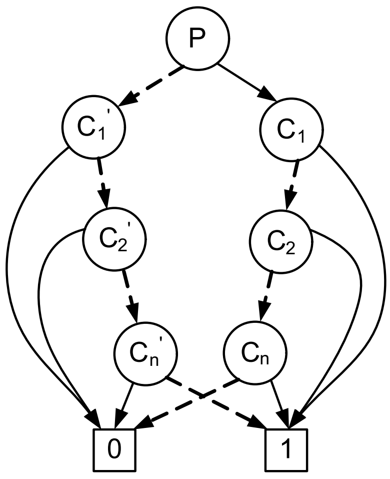
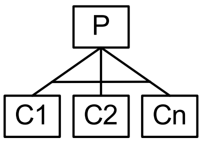
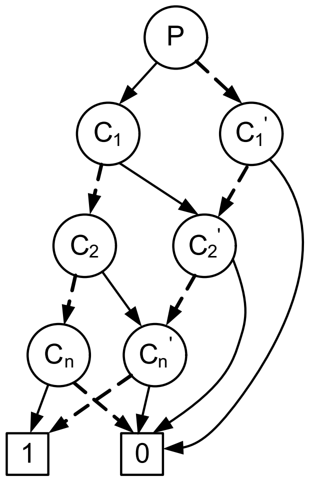
RELATION		Formula	BDD
OR		$P \leftrightarrow (C_1 \vee C_2 \vee \ldots \vee C_n)$	
ALTERNATIVE		$(C_1 \leftrightarrow (\neg C_2 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge$ $(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_{n-1} \wedge P))$	

Figure 9.4: *Mapping* or *and* alternative *relationships to BDD.*

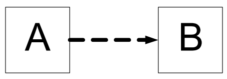
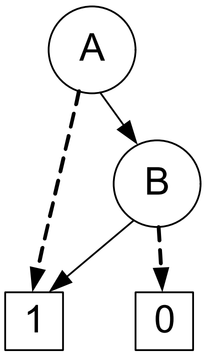
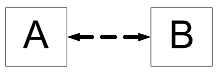
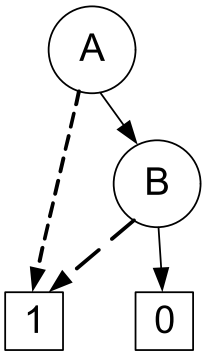
RELATION		**Formula**	**BDD**
REQUIRES		$A \rightarrow B$	
EXCLUDES		$\neg(A \wedge B)$	

Figure 9.5: *Mapping* requires *and* excludes *relationships to BDD.*

9.5 Performance comparison

When we first implemented our mapping, we found that there were some operations that did not scale when the number of features in the feature model grew. Therefore, we decided to set up some performance comparisons. Firstly, we compared two general CSP solvers. Afterwards, we compared one of these general CSP solvers with SAT and BDD solvers. Following, we present the results of this comparison.

9.5.1 General CSP solvers comparison

JaCoP and Choco solvers were compared in terms of performance. With the following test we intended to show which general CSP solver provides the best performance in the automated analyses of feature models. In addition, we studied the robustness and the areas of vulnerability of each solver. In order to evaluate both solvers we used five feature models (we call them test cases). Three of them represent small and medium size real systems, meanwhile the larger two were generated randomly for this test. After formulating each one as a CSP in both platforms, we proceeded with the execution. Table §9.1 summarises the characteristics of the feature models used. Feature Model 1 is a simple feature model representing a home integration system. It was presented in [23]. Feature Model 2 is the one of Figure §2.8 (pag. 34) which represents a collaborative web based system. Feature Model 3 is a medium size feature model of a flight booking system based on the work done by Díaz *et al.*[50, 51]. Finally, we generated two larger feature models randomly (Feature Models 4 and 5) with a double aim: representing more complex systems with a greater number of features and dependencies, and evaluating the solvers performance in more difficult situations. We considered it was necessary to compare the performance with small, medium and larger feature models in order to evaluate solver performance results in different situations.

The process for generating a feature model randomly is based on a recursive method that has four input parameters: height levels, maximum number of children relationships for a node, maximum number of elements in a *set* relationship and number of dependencies. Firstly, features and their relationships are generated using random values. Secondly, cross-tree constraints are created by taking pairs of features randomly and establishing a random dependency (*requires* or *excludes*) between them. We made sure not to generate misconceptions (e.g. a child depends on a parent).

For these test cases two operations were performed: *i*) finding one configuration that would satisfy all the constraints, i.e., a product and *ii*) finding

Feature Model	N. of Features	N. of Dep
1	15	0
2	14	2
3	26	0
4	40	14
5	52	28

Table 9.1: *Feature models used to test general CSP solvers.*

the total number of configurations of a given feature model. The first is the simplest operation while the second is the most difficult one in terms of performance because it is necessary to retrieve all possible combinations.

The comparison focused on the data obtained from several executions in order to avoid as much exogenous interferences as possible. In every test case, the total number of executions to calculate the average time was ten. The data extracted from the tests was:

- Number of features in the first solution obtained by each solver.
- Average execution time to obtain one solution (measured in milliseconds).
- Total number of solutions, that is, the potential number of products represented by the feature model.
- Average execution time to obtain the number of solutions (measured in milliseconds).

In order to evaluate the implementation, we measured its performance and effectiveness. We implemented the solution using Java 1.5.0_04. We ran the tests on a WINDOWS XP PROFESSIONAL machine equipped with a 3.2Ghz Intel Pentium IV microprocessor and 1024 MB of DDR 166Mhz RAM memory.

The Results

The performance comparison revealed some interesting results (see Figures §9.6, §9.7 and §9.8). The first evidence we should mention is that JaCoP was on average 54% faster than Choco in finding one solution. It is important to observe that our approach was feasible in these test cases because the necessary time to obtain a response is really low (35 milliseconds in the worst case).

	JACOP / CHOCO					
Experiment	**Features in Sol.**	**Time one Sol.**		**Nº Solutions**	**Time all Sol.**	
		JACOP	**CHOCO**		**JACOP**	**CHOCO**
1	7	9,9	18,8	32	37,5	45,5
2	8	9,4	22,7	68	64,4	81,3
3	13	12	24	512	225,6	265,3
4	19	20,2	34,9	34560	5619	2203,3
5	19	24,4	35,8	61440	15390,8	4817,6

Figure 9.6: *Results of JaCoP and Choco solvers comparison.*

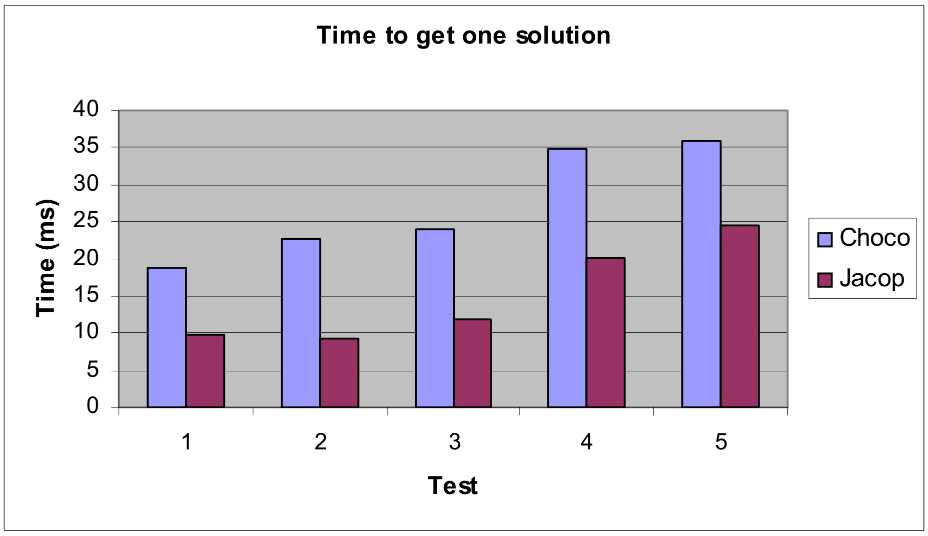


Figure 9.7: *Comparing JaCoP and Choco getting one solution.*

However, while JaCoP was much faster than Choco in finding the total number of solutions in small CSPs, JaCoP seems to be noticeably slower than

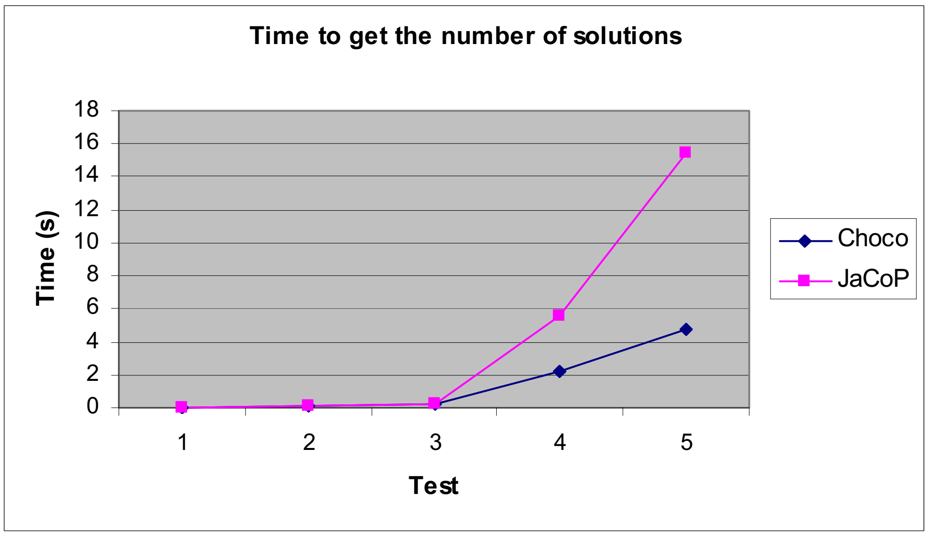


Figure 9.8: *Comparing JaCoP and Choco getting the number of solutions.*

Choco in the big ones (see Figure §9.8). This curious result probably depends on how each solver was used to obtain the number of solutions. Choco has a simple method to know the number of solutions of a concrete problem (`Solver.getNbSolutions()`), while JaCoP implementation needed to find all the solutions first and count them afterwards. This simple variation implies a very important difference in performance. For instance, in test case 5, JaCoP needed to create 61440 `ArrayLists` and fill all of them with all the solutions which produces a great time loss. On the other hand, Choco did not have this weakness because, although Choco does actually find all solutions, its method to find the number of solutions only returned five of them to avoid memory deficit problems. If the user wants to obtain the other solutions he only has to make a simple iteration and take them one by one. In the three smaller test cases, JaCoP was faster than Choco so we presume that this trend would continue if JaCoP optimized this aspect. In test 5, we performed a comparison to find and return all the solutions in both solvers, that is, not only to find the number of solutions but the solutions themselves. The result was decisive: Choco required over a minute to perform this task, proving to be slower than JaCoP in this situation.

Although memory usage was not a relevant data in our tests we noticed that in general Choco uses more memory than JaCoP; however there is not a remarkable difference between both solvers.

Finally, we identified some interesting characteristic in both solvers. Firstly, JaCoP allows the user to obtain easily from executions more interesting information than Choco such as the number of backtracks of a search or the number of decisions taken to find a solution. In second place, we found a worrying bug when working with big problems in Choco. In most cases, executions of CSPs representing big feature models generated an exception (`choco.bool.BinConjunction`) which imposes an important limitation to Choco and limited us in the number of features of the feature models for the test cases.

9.5.2 Multisolver comparison

In this Section we present the results of another comparison we performed. We focused on a performance comparison of three different type of solvers working with CSP, SAT and BDD in order to test how these representations can influence in the automated analysis of feature models. The comparison results were obtained from the execution of a number of feature models translated into CSP, BDD and SAT in three solvers presented previously: JaCoP (general CSP solver), JavaBDD (BDD solver) and Sat4j (SAT solver).

N. of Features	N. of instances	% of Cross-tree constraints	Total N. of test cases
[50-100)	50	[0%-25%]	931
[100-150)	50	[0%-25%]	1566
[150-200)	50	[0%-25%]	2276
[200-300]	50	[0%-25%]	2582

Table 9.2: *Instances generated for multisolver comparison.*

We used four groups of 50 randomly generated feature models. Each group included feature models with a number of features in a specific range ([50-100), [100-150), [150-200) and [200-300)) with a double aim: test the performance of small, medium and larger instances and working out averages from the results in order to avoid as much exogenous interferences as possible.

After formulating each feature model as a CSP, BDD and SAT, we proceeded with the execution. Each feature model was executed several times increasing the number of cross–tree constraints from 0 up to 25% of the number of features in the feature model in step of 1%. Notice that there is actually a bigger number of different test cases generated from every base feature model.

Column “Total N. of test cases” of Table §9.2 presents the total number of test cases in every range.

We increased the number of cross–tree constraints in order to know how these constraints influenced in the performance. Cross-tree constraints were added randomly as well, but checking that the same feature could not appear in more than one cross–tree constraint and that a feature can not have a cross–tree constraint with any of its ancestors. Averages were obtained from every test case in each range with the same percentage of cross–tree constraints. Table §9.2 summarises the characteristics of the comparison.

For the comparison two operations were performed: *i*) finding out if a model is void, i.e., it has at least one solution and *ii*) finding the total number of products of a given feature model. The first one is the simplest operation while the second is the hardest one in terms of performance because it is necessary to work out the total number of possible combinations. The data extracted from the tests were:

- Average memory used by the logic representation of the feature model (measured in Kilobytes)
- Average execution time to find one solution (measured in milliseconds).
- Total number of solutions, i.e. the potential number of products represented in the feature model.
- Average execution time to obtain the number of solutions (measured in milliseconds).

In order to evaluate the implementation, we measured its performance and effectiveness. We implemented the solution using Java 1.5.0_04. We ran our tests on a WINDOWS XP PROFESSIONAL SP2 machine equipped with a 3Ghz Intel Pentium IV microprocessor and 512 MB of DDR RAM memory.

The Results

The performances comparison revealed some interesting results. The first evidence was that JavaBDD is on average 96% faster than JaCoP and 75% faster than Sat4j finding one solution, i.e. determining if a feature model is void. However, JavaBDD revealed a memory usage on average 928% higher than JaCoP and 1672% higher than Sat4j. On the other hand, although JaCoP and Sat4j showed a similar memory usage, SAT representation showed better results in both aspects, memory and especially in time. The performance of the solvers was similar in the four groups of test cases. Figures §9.9 and §9.10 presents the results for the group of feature models with a number of features between 100 and 150.

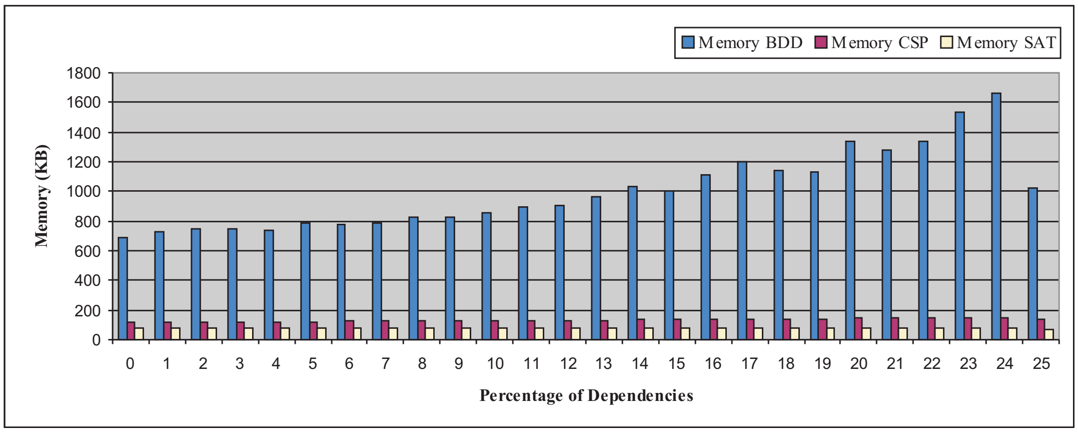


Figure 9.9: *Memory usage of SAT, BDD and CSP solvers.*

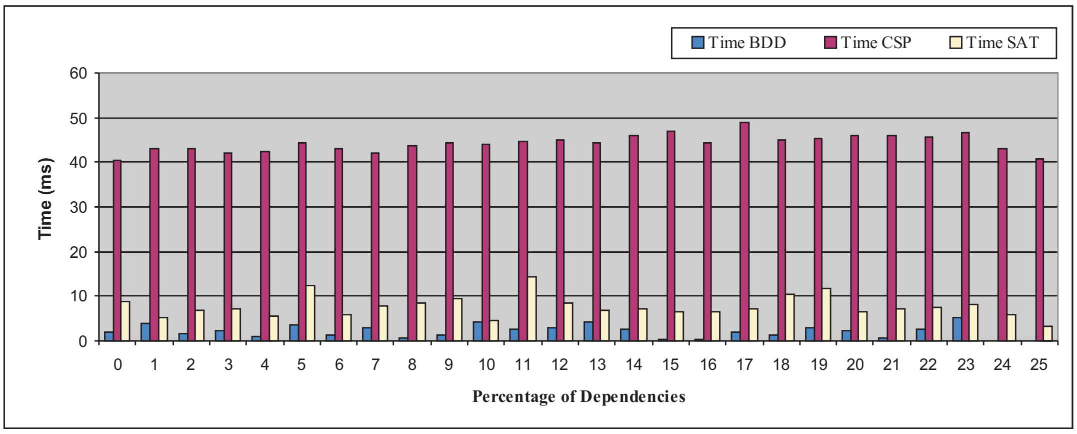


Figure 9.10: *Average time to get one solution of SAT, BDD and CSP Solvers.*

In fact, Figure §9.9 can be confusing in the sense that in the worst case the memory usage is insignificant (in the order of 2 Mb) but this behaviour seems to be exponential with the number of features and dependencies (cross–tree constraints). For instance, in the range of (200-300) features, we found some cases where the memory used by the solver was around 300 Mb. We think that in bigger feature models (e.g. 1000 features) this can be even a bigger problem. What Figure §9.9 tries to stress is the difference in the use of memory of the three solvers.

The results obtained from finding the total number of products of a given feature model showed a great superiority of JavaBDD. While JaCoP and Sat4j were computationally incapable of performing that operation in a reasonable time in most of the cases, JavaBDD lasted 5312 ms to work out the 7.77×10^{34} solutions of the worst case.

Finally, we found some unexpected results or outliers in the data obtained from the tests with JaCoP and JavaBDD. On the one hand, JaCoP presented in a few consecutive executions a huge number of backtracks and consequently a great time penalty. On the other hand, JavaBDD revealed in a few tests a huge memory usage which seemed to increase exponentially with the number of dependencies. We are investigating the possible causes of these behaviours [20].

9.5.3 Discussion

The great superiority of JavaBDD on finding the total number of solutions is because for calculating the number of solutions, in general, CSP and SAT solvers have to retrieve all the solutions (which is a #P-complete problem [80]) meanwhile BDD solvers use efficient graph algorithms to calculate the total number of solutions without the need of calculating all the solutions. The huge memory usage of BDD solvers depends on the variable ordering for representing the BDD. The size of BDDs can be reduced with a good variable ordering, however, calculating the best variable ordering is a known NP-hard problem [35].

To the best of our knowledge, feature attributes such as time, cost, versions of module, and so on, can easily expressed using CSP solvers. By contrast, we are not aware of any result that allows to easily express feature attributes in a BDD or SAT representation, and hence BDD and SAT solvers cannot be used to find the best product for a feature model that maximises or minimizes some objective.

As a result of the test, we claim that there is not an optimum representation for all the possible operations that can be performed on feature models. Therefore we propose a multisolver implementation of the implementation layer of FAMA.

9.6 Summary

In this chapter, we have described a proof of concepts implementation of FAMA. This is the implementation layer of FAMA. We have described the solvers we used in the implementation; later we have provided a mapping from abstract CSP defined in previous chapter to these concrete solvers. We have also showed two performance test we accomplished. We concluded that there is not an optimal representation for all operations so we propose the use of a multisolver approach to profit the best of every solver.

We have published part of these results in some papers, namely: we presented the translation of feature models into CSPs in [16, 22–24]. We presented the results of the tests with two CSP solvers in [27] and the test case with three type of solvers in [26].

Part IV
Final remarks

Chapter 10

Conclusions and future work

My interest is in the future because
I am going to spend the rest of my life there

Charles F. Kettering, 1876–1958
Engineer

10.1 Conclusions

In this dissertation we have shown that:

> *A framework based on rigorous definitions, with a high level of abstraction, supporting attributed features and supporting the use of different solvers can provide practical means of analysing software product lines in general and feature models in particular.*

Software product lines are a consequence of the mass customization economical paradigm shift. Feature models are models used to represent all possible products of software product lines in a compact representation. They were proposed back in 1990 and are recognized by some authors as one of the most important contributions in the field of software product line engineering. The automated analysis of software product lines deals with the computer–aided extraction of information from a software product line. The automated analysis of software product lines in general and feature models in particular is a key task in a software product line approach and is an ongoing research area. The importance of the automated analysis of feature models resides in the fact that this analysis is done without the participation of other software artefacts like code, documents and so on, i.e. the analysis is done at a very high level of abstraction allowing to detect errors or properties in a very early stage of development.

However, current proposals for the automated analysis of software product lines present some problems. They lack abstraction, in the sense that they are only focussed on a special case of feature models. They lack formal semantics, which can lead to misinterpretations. They do not support the analysis of feature models taking into account feature attributes and finally, they do not support more than one type of solver in the implementation of the analysis.

Our goal in this thesis has been to devise a new framework called FAMA to overcome the aforementioned problems. FAMA is a four–layers framework. It is abstract, it supports the analysis of extended feature models, it is rigorously defined and finally it support multiple solvers in the implementation layer. FAMA can be extended to support more operations of analysis, it can be extended to support other models than feature models and can be extended to support other solvers.

10.2 Discussion, limitations and extensions

Paraphrasing Shakespeare in *The Merchant of Venice* it would be fair to say that *"All that glitters is not gold"*. Following, we discuss some of the main decisions we have made in this dissertation highlighting its main limitations and possible extensions.

- *Is a formal language suitable for specifying a framework for the analysis of feature models?* We have used a formal specification language such as Z for the specification of our framework. This has brought us important benefits in terms of rigour and clarity in the specification, which was the goal we pursued. However, using formal languages sometimes has some drawbacks too. On the one hand, the common view is that formal mathematical notations are out of reach of common software engineers. This can be a negative factor in submitting our proposal to a broader audience. On the other hand, we appeal to Bowen and Hinchey's "Ten commandments for formal methods" [33] to agree that for using formal methods *"Tho shalt have a formal methods guru on call"* which is not always easy. For instance, in the development of the FAMA Eclipse plug–in (see Appendix §A), we have not thoroughly used the rigorous definitions of the specification presented in this dissertation so that the tool is not an accurate reflection of the rigours specification. The reason is that developers of the prototype did not understand formal methods as well as UML models and metamodels.

Conclusion: if we went back to the beginning and we had to decide whether to use a formal language or not, we would once again decide to use it for our dissertation.

Extension: Revise our tool in an iterative process to completely match with our specification or change the specification if needed according to tool needs.

- *Are feature attributes completely contemplated in FAMA?* Although we have claimed the need of including attributes and relationships among attributes in feature models, in our rigorous framework we have only specified attributes of features as attribute–value pairs. We have not specified the type of possible relationships among attributes and how these relationships can influence in the analysis operations. For instance, if we have a relationship like "X *requires* Y.attribute > Z" then, the number of possible products of the feature model can be reduced and the number of products operation would need to be revised among others.

  Extension: Including attribute relationships in FAMA and studying the possible revision on the operations.

- *Are all operations contemplated in FAMA?* Although we have listed some operations in Chapter §3, we did not deal with modification operations. Extending FAMA to support modification operations over feature models to confront staged configuration processes is also something that has to be studied. In this context, features would not have only two states (selected, deselected) but several (selected, deselected, decided, undecided and so on) which may make the specification somewhat more complicated.

  Extension: Extending FAMA with modification operations.

- *Are CSPs the best operational support?* We have chosen constraint programming (based on CSPs) as the operational paradigm. These allowed us to have a higher level of abstraction in the specification since it is always possible to translate any CSP into a propositional—based specification [32]. The language of CSP solvers is often more succinct than the one of propositional–based solvers (BDD or SAT). In addition, it is believed in the constraint programming community (although not proved yet in general) that CSP solvers can be better than propositional–based solvers for non-boolean problems. However, having chosen this paradigm has some limitations. On the one hand, it reduces our analysis capabilities in terms of ontological or fuzzy relations. On the other hand, it is well–known that CSPs are in general NP–complete problems. We have proved that determining iff a feature model is void is an NP–complete problem as well. However, we have neither made a complete computational complexity study of the operations of analysis, nor performed an empirical feasibility study in large–scale real software product lines. Finally, in our proof of concept, we have only introduced discrete domains for attributes such as integers or sets.

Conclusion: At the time of writing, constraint programming paradigm seems to be the most appropriate operational paradigm for the analysis of feature models.

Extensions:

◇ Extending FAMA with description logic reasoners for ontological analyses and fuzzy logic reasoners for fuzzy analyses.

◇ Performing computational complexity and feasibility studies of FAMA operations.

◇ Extending FAMA implementation to continuous domains.

10.3 Other future work

The results in this dissertation can be seen as the conclusion of a first lap. However, there is still a long way to go. There are many issues that remain open or to be improved. Following, we revise some of the topics of our contributions (see Section §1.2.1 on page 8) to motivate some other possible future work.

- *Tool support*. FAMA Eclipse plug–in is a tool that is already functional but there is still a large amount of work to be done in this area. On the one hand, Are boxes and lines the best way of representing feature models? There is an interest in the community concerning these topics [1]. On the other hand, is FAMA–EP itself a software product line? FAMA can be seen as a software product line itself.

Future work:

◇ Studying visualization of feature models.

◇ Feature refactoring of FAMA to set up a software product line of product lines analysis tools.

- *Survey and research agenda.* The state of the art and research agenda has to be reviewed from time to time in order to summarize the progress in the area. In addition, more rigorous surveys are still missing and this is a field that we plan to explore as well.

  Future work: Revise the state of the art and the challenges ahead on the analysis of feature models.

- *Web services and software product lines* (WS-SPL). Gathering web service development and software product lines is not a trivial problem and there are still plenty of open issues in this area. This is already being studied in Sergio Segura's research work and is one of the main topics of WEBFACTORIES, the research project in which we are currently involved.

  Future work: Studying how FAMA can be used in real developments in conjunction to feature oriented programming approaches using its automated analysis capabilities in the production of software product lines based on web services.

- *Industrial experiences.* We have tried to apply in our research work an action research method [7, 17, 21] in which, instead of inventing new problems, we try to extract them from industry and provide valid solutions. We have some tasks in WEBFACTORIES to bridge this gap. In addition, Jorge Müller and Manuel Nieto's research work will deal with these issues.

  Future work: Applying our ideas in real settings to obtain feedback, and find for more problems to be solved.

- *Error analysis.* We have pioneered the use of theory of diagnosis for error analysis in feature models. This is something that is being already studied in Pablo Trinidad's thesis that follows up our work on this dissertation.

  Future work: Consolidating theory of diagnosis for error analysis of feature models.

- *MDD and transformation of feature models.* Just as feature models can be analysed, they can be also transformed. We have no results in this field yet but we believe that this is an open research area that has to be investigated to leverage automated analysis of feature models. This is also something that is being studied in the frame of WEBFACTORIES.

  Future work: Mapping feature selections in a feature model into other development artifacts (requirements, architecture, processes, code modules, test cases, documentation, etc.) and verifying that other program representations are consistent with their feature model.

10.4 A last phrase to conclude

As a final conclusion, we would like to paraphrase the Greek philosopher Socrates saying that:

After years of hard work, *we know nothing except the fact of our ignorance.*

Part V
Appendices

Appendix A

FAMA Eclipse plug–in

A picture is worth a thousand words.
An interface is worth a thousand pictures.

Ben Shneiderman, 1947–
Usamerican computer scientist.

The automated analysis of feature models is recognized as one of the key challenges for automated software development in the context of software product lines. However, after years of research only a few ad-hoc proposals have been presented in such area and the tool support demanded by the software product lines community is still insufficient. In previous chapters we showed how the selection of a logic representation and a solver to handle analysis on feature models can have a remarkable impact in the performance of the analysis process. In this chapter we present a first implementation of FAMA as an Eclipse plug–in that integrates some of the most commonly used logic representations and solvers proposed in the literature. To the best of our knowledge, FAMA Eclipse plug–in is the first tool integrating different solvers for the automated analyses of feature models.

This chapter is structured as follows: in Section §A.1 we introduce the chapter; Section §A.2 provides with a general overview of FAMA Eclipse plug-in. In Section §A.3, we describe the main components of the FAMA–EP architecture. A summary of some related tools is introduced in Section §A.4. Finally we summarize our conclusions and describe our future work in Section §A.5.

A.1 Introduction

As we have described during this dissertation, feature modelling is a common mechanism for expressing a set of products of a software product lines in terms of features in a single model being a feature an increment in product functionality.

We have showed all along this dissertation that the automated analysis of feature models is recognized in the literature as an important challenge in software product line research [9, 11].

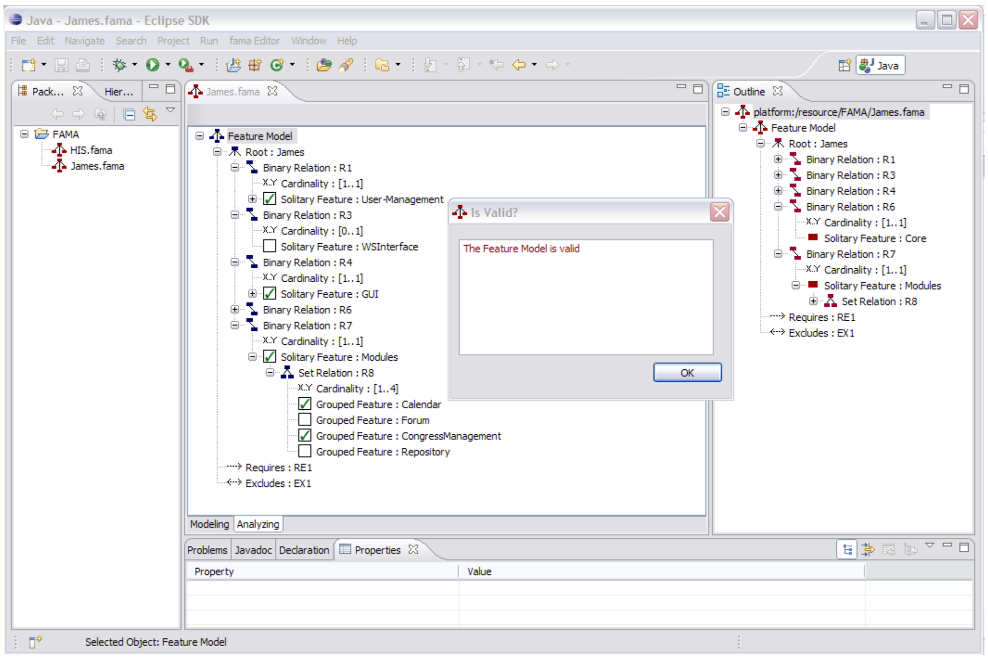
The automated analyses of feature models is usually performed in two steps: *i*) The feature model is translated into a certain logic representation *ii*) off-the-shelf solvers are used to extract information from the result of the previous translation such as the number of possible products of the feature model, all the products following a criteria, finding the minimum cost configuration, etc [25] (see Section §6.3, page 78). We have introduced how the use of different solvers [27] and logic representations [26] can have an important effect in the time and memory performance of the analysis process (see Chapter §9). We also showed that there is not an optimum logic representation and solver for all the operations that can be performed on feature models.

In this appendix, we present a first prototype implementation of *FAMA Eclipse Plug–in* (FAMA–EP). FAMA–EP is an extensible framework for the automated analysis of feature models. FAMA–EP can be configured to select automatically in execution time the most efficient of the available solvers according to the operation requested by the user. The current implementation of FAMA–EP integrates three of the most promising logic representations proposed in the area of the automated analysis of feature models: CSP, SAT and BDD, but more solvers can be added if needed.

A.2 FAMA–EP general overview

FAMA–EP has been implemented as a tool for the edition and analysis of feature models. FAMA–EP supports cardinality-based feature modelling (that includes traditional feature models, e.g FODA, Feature–RSEB, and so on), export/import of feature models from XML and XMI (XML Metadata Interchange) and analysis operations on feature models. In order to make our tool multiplatform and easy to access, we implemented FAMA–EP as a plug–in for the Eclipse Platform[†1] (see Figure §A.1).

[†1] `www.eclipse.org`


Java - James.fama - Eclipse SDK
File Edit Navigate Search Project Run fama Editor Window Help
Java
Pack... Hier...
FAMA
HIS.fama
James.fama
James.fama
Outline
Is Valid?
OK
Modeling Analyzing
Problems Javadoc Declaration Properties
Property Value

```
Feature Model
  Root : James
    Binary Relation : R1
      X..Y Cardinality : [1..1]
      ✓ Solitary Feature : User-Management
    Binary Relation : R3
      X..Y Cardinality : [0..1]
      ☐ Solitary Feature : WSInterface
    Binary Relation : R4
      X..Y Cardinality : [1..1]
      ✓ Solitary Feature : GUI
    Binary Relation : R6
    Binary Relation : R7
      X..Y Cardinality : [1..1]
      ✓ Solitary Feature : Modules
        Set Relation : R8
          X..Y Cardinality : [1..4]
          ✓ Grouped Feature : Calendar
          ☐ Grouped Feature : Forum
          ✓ Grouped Feature : CongressManagement
          ☐ Grouped Feature : Repository
  Requires : RE1
  Excludes : EX1
```

The Feature Model is valid

```
platform:/resource/FAMA/James.fama
  Feature Model
    Root : James
      Binary Relation : R1
      Binary Relation : R3
      Binary Relation : R4
      Binary Relation : R6
        X..Y Cardinality : [1..1]
        Solitary Feature : Core
      Binary Relation : R7
        X..Y Cardinality : [1..1]
        Solitary Feature : Modules
          Set Relation : R8
    Requires : RE1
    Excludes : EX1
```

Selected Object: Feature Model

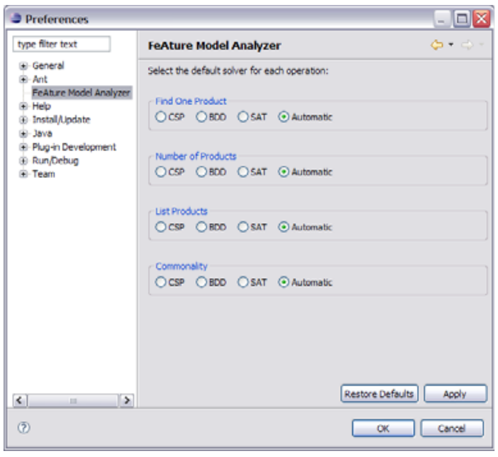
Figure A.1: *Screenshot of the FAMA–EP.*

FAMA–EP offers two main functionalities: visual model edition / creation and automated model analysis. Once the user has created or imported (from XML / XMI) a feature model, the analysis capability can be used. Most of the operation identified on feature models [25] are being currently implemented. At the moment of writing this chapter the operations fully supported by FAMA–EP are:

- Finding out if a feature model is void, i.e. it exists at least one product satisfying all the constraints.
- Finding the total number of possible products of a feature model (number of products).
- List all the possible products of a feature model (list of products).
- Calculate the commonality of a feature.

FAMA–EP integrates different solvers in order to combine the best of all of them in terms of performance. The actual version of the framework integrates JaCoP as general CSP solver, SAT4J as SAT solver and JavaBDD as BDD solver to perform the analysis tasks. However, an unlimited number of new analysis operations and solvers could be added.

One of the advantages of FAMA–EP is the ability to select automatically, in execution time, the most efficient solver according to the operation requested by the user. Therefore, if the user asks for the number of possible combinations of features of a feature model the framework will select automatically the BDD solver to get it (the most efficient known approach for this operation in terms of performance). The automated selection of a solver is based on the value of some configuration parameters establishing the priority between the available solvers for each operation. The values of these parameters were set according to the results from the performance test of the solvers integrated in the framework that was presented in previous chapters. However, it is also possible to configure this values for each operation by the user (see Figure §A.2).



FeAture Model Analyzer

Select the default solver for each operation:

Find One Product
CSP BDD SAT [✓ Automatic]

Number of Products
CSP BDD SAT [✓ Automatic]

List Products
CSP BDD SAT [✓ Automatic]

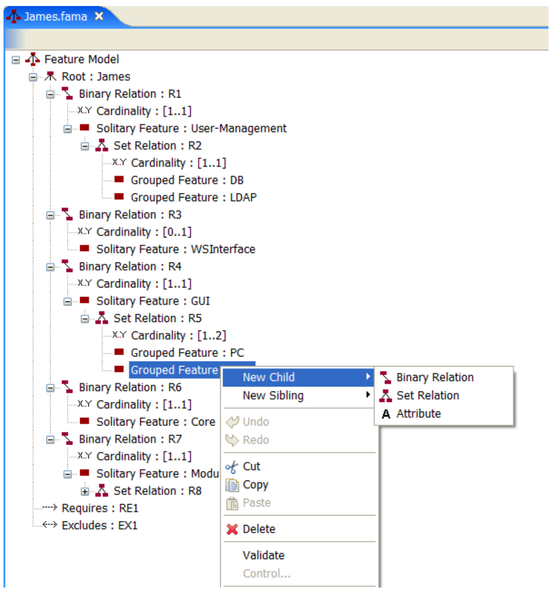
Commonality
CSP BDD SAT [✓ Automatic]

Figure A.2: *Preferences page of the FAMA–EP.*

A.3 The FAMA Architecture

FAMA–EP is divided in two main components: the *analysis engine* and the *visual editor*. To interact with the automated analysis framework, an user interface has been implemented as an extensible Eclipse plug-in. The user interface consists of two customized tree views: the *modeling view* (Figure §A.3) that allows the visual edition of feature models, and the *analysis view* (Figure §A.4).

The *analysis engine* is responsible for performing the analysis operations requested by the users, and the *visual editor* offers a graphical user interface for



```
James.fama
Feature Model
  Root : James
    Binary Relation : R1
      Cardinality : [1..1]
      Solitary Feature : User-Management
        Set Relation : R2
          Cardinality : [1..1]
          Grouped Feature : DB
          Grouped Feature : LDAP
    Binary Relation : R3
      Cardinality : [0..1]
      Solitary Feature : WSInterface
    Binary Relation : R4
      Cardinality : [1..1]
      Solitary Feature : GUI
        Set Relation : R5
          Cardinality : [1..2]
          Grouped Feature : PC
          Grouped Feature
    Binary Relation : R6
      Cardinality : [1..1]
      Solitary Feature : Core
    Binary Relation : R7
      Cardinality : [1..1]
      Solitary Feature : Modu
        Set Relation : R8
  Requires : RE1
  Excludes : EX1
```

Figure A.3: *Modeling view of the FAMA–EP.*

feature modelling and interacts with the analysis engine. The communication between such modules is minimum in order to reduce the coupling among them. In fact, the analysis engine can be used as an off-the-shelf tool for the integration of automated analysis of feature models in other tools. Next we show in detail the structure of each module separately.

A.3.1 The analysis engine

Figure §A.5 shows the UML component diagram of the FAMA analysis engine.

The analysis engine relies on the concept of *questions*. An analysis operation is carried out by a question that is asked by the central component of the architecture which is the *QuestionTrader*. For each operation that the engine supports, a question interface is defined.

On the other hand, for each logical paradigm at least one solver is used to implement it. Each solver (called reasoner) implements the *SolverReasoner* interface and uses an off–the shelf solver to provide the analysis functionality. A solver can implement as much operations as desired from the operations available in *QuestionTrader*.

Among the responsibilities of *QuestionTrader* we find:

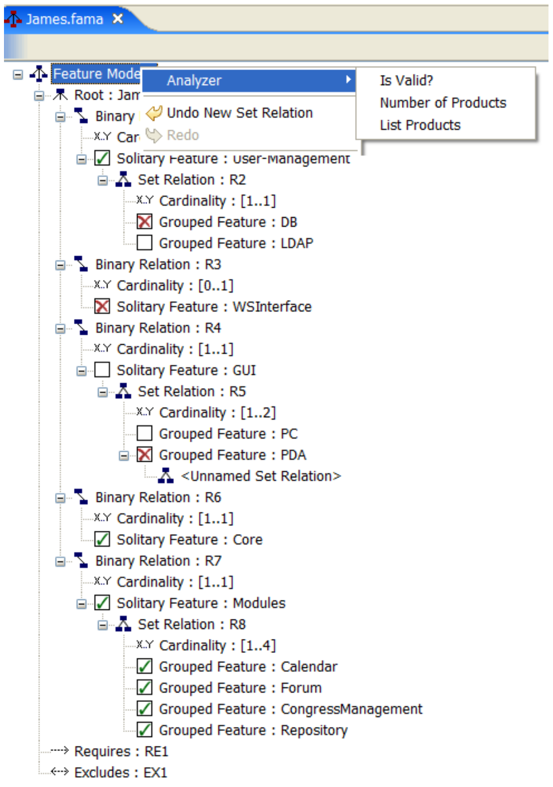

```
James.fama ✕
Feature Mode   Analyzer ▸            Is Valid?
  Root : Jam                         Number of Products
    Binary    Undo New Set Relation  List Products
      X.Y Car  Redo
      ☑ Solitary Feature : User-Management
        Set Relation : R2
          X.Y Cardinality : [1..1]
          ☒ Grouped Feature : DB
          ☐ Grouped Feature : LDAP
    Binary Relation : R3
      X.Y Cardinality : [0..1]
      ☒ Solitary Feature : WSInterface
    Binary Relation : R4
      X.Y Cardinality : [1..1]
      ☐ Solitary Feature : GUI
        Set Relation : R5
          X.Y Cardinality : [1..2]
          ☐ Grouped Feature : PC
          ☒ Grouped Feature : PDA
            <Unnamed Set Relation>
    Binary Relation : R6
      X.Y Cardinality : [1..1]
      ☑ Solitary Feature : Core
    Binary Relation : R7
      X.Y Cardinality : [1..1]
      ☑ Solitary Feature : Modules
        Set Relation : R8
          X.Y Cardinality : [1..4]
          ☑ Grouped Feature : Calendar
          ☑ Grouped Feature : Forum
          ☑ Grouped Feature : CongressManagement
          ☑ Grouped Feature : Repository
  Requires : RE1
  Excludes : EX1
```

Figure A.4: *Analysis view of the FAMA–EP.*

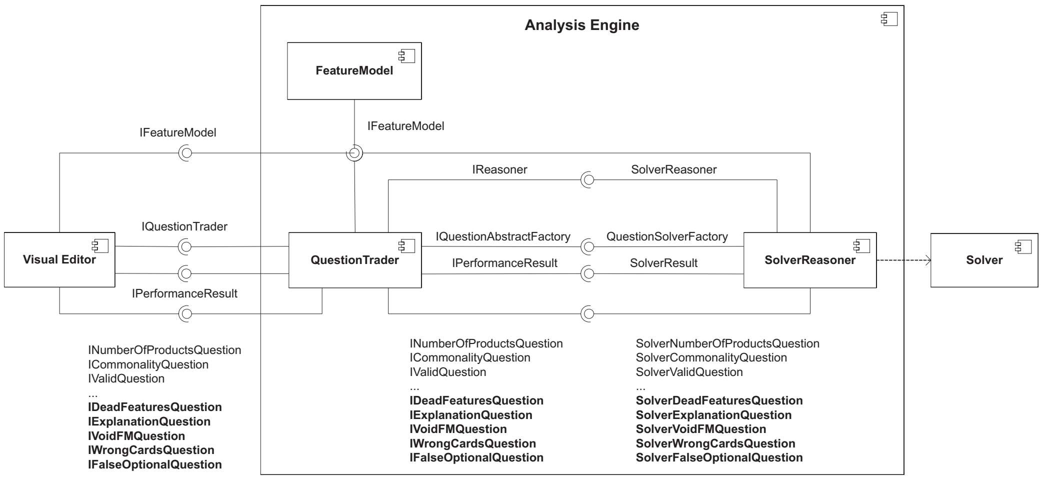


Figure A.5: *UML component diagram of the FAMA Analysis Engine.*

- Containing and registering all the available reasoners, i.e. giving support to multiple paradigms.

- Knowing about the questions that are supported and which reasoners are able to answer them. It allows to extend the functionality by adding new questions.

- Selecting the reasoner that performs better to answer a question that is performed by the user.

- Measuring the performance a reasoner gives when answering a question.

- Reporting to a solver the changes on the feature model.

QuestionTrader component concentrates the main responsibilities of the module and new solvers and questions can be added easily by implementing the interfaces provided and registering them on this component. It eases the usage of multiple paradigms to the user, as only the operation to perform on a feature model must be pointed as a question and *QuestionTrader* will automatically select a reasoner and will answer the question.

A.3.2 The visual editor

The graphical user interface of FAMA is provided by a customized and extended visual editor (Figure §A.6) implemented using the Eclipse Modeling Framework (EMF)[†2]. EMF is a modeling framework and code generation facility for building tools based on a structured data model. In our case, the data model is the feature model metamodel that has been used for the analysis engine with some extensions to support data that is needed for edition. The data model is described in XMI, being EMF able to automatically generate a default visual editor that contains three components:

- *ExtendedFM component*: a set of classes that represent the elements in the feature model.

- *EditableFM component*: a set of adapter classes that enable models edition interacting with *ExtendedFM* component.

- *Editor component*: a basic user interface to edit the feature model.

[†2]http://www.eclipse.org/emf

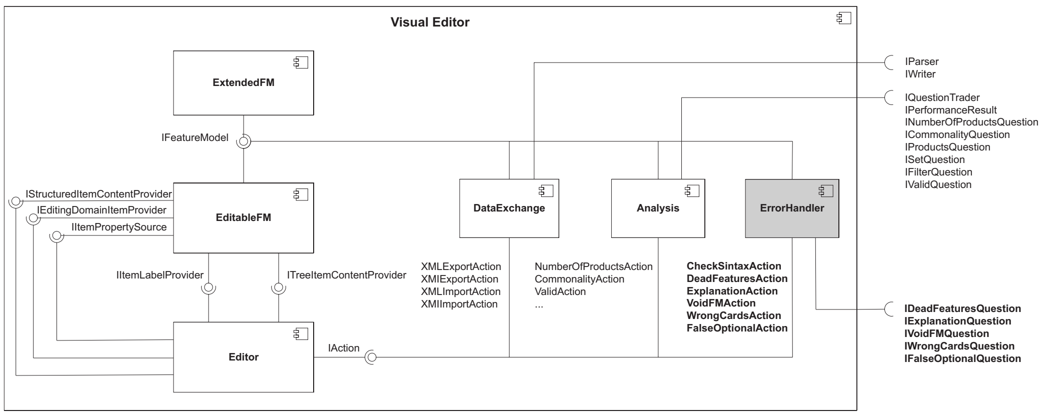


Figure A.6: *UML component diagram of the FAMA Visual Editor.*

The functionalities provided by the visual editor except those regarding to feature modeling that are implemented in the *editor* component, are described as *actions*. The actions are grouped in different components according to the functionality they provide. Currently we have defined three action components:

- *Analysis actions*: Actions involving analysis operations on feature models. They interact with the analysis engine by means of the interfaces provided by the QuestionTrader component.
- *Data exchange actions*: actions to export/import the feature models into XML or XMI files so they can be stored or restored later. They can be used to interoperate with other applications.
- *Error handling actions*: actions to implement error analysis (this component is under development).

A.4 Similar tools

The Feature Model Plugin[†3] (FMP) [48] has also been implemented as an Eclipse plug–in. It supports cardinality-based feature modelling, specialization of feature diagrams and configuration based on feature diagrams. FMP uses a BDD solver to work out the number of possible combination of features in a feature model. It also supports the use of filters. FMP is becoming

[†3]`http://gp.uwaterloo.ca/fmp/`

a mature tool with interesting extensions but it does not have the analysis of feature models between its main goals. It does not support attributed feature models and do not include more than one solver for the analysis.

XFeature[†4] [37] is an XML-based feature modelling tool also implemented as an Eclipse plug–in. XFeature supports the modelling of product families and applications instantiated from them. This tool does not support the automated analysis of feature models.

CaptainFeature[†5] is a feature modelling tool using the FODA notation to render feature diagrams. It also includes an integrated configurator to specialize the created feature diagrams. CaptainFeature does not support the automated analysis of feature models.

Requiline[†6] [100] is defined as a requirement engineering tool for the efficient management of software product lines. From the edition of a group of features and requirements Requiline derives product configurations. It also includes a consistency checker and an query and XML interface. Apart from the consistency checking, RequiLine does not perform any of the others analysis operation identified on feature models [25].

Pure::Variants[†7] is a commercial tool supporting feature modelling and configuration. Current version does not support cardinalities. Pure::Variants supports basic analysis operations through a Prolog-based constraint solver.

The *AHEAD Tool Suite*[†8] (ATS) [10] is a set of tools for product-line development that support feature modularizations and their compositions. AHEAD can perform certain analysis operations on feature models by means of SAT solvers [9]. It does not include feature models attributes in the analysis. Feature models are saved as grammars and no XML representation is provided.

Table §A.1 summarizes the exposed proposals.

A.5 Summary

In this chapter we introduced the first prototype implementation of FAMA–EP. We introduced the logic representations currently used in the framework

[†4]`www.pnp-software.com/XFeature/`
[†5]`https://sourceforge.net/projects/captainfeature/`
[†6]`www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline`
[†7]`www.pure-systems.com`
[†8]`www.cs.utexas.edu/users/schwartz/ATS.html`

	FMP	XFeature	CaptainFeature	RequiLine	Pure::Variants	AHEAD Tool Suite	FAMA–EP
CSP	-	-	-	-	-	-	+
SAT	-	-	-	-	-	+	+
BDD	+	-	-	-	-	-	+
Prolog	-	-	-	-	+	-	-
Multi Solver	-	-	-	-	-	-	+
Basic FMs	+	+	+	+	+	+	+
Cardinality-based FMs	+	+	-	-	-	-	+
XML / XMI	+	+	-	+	+	-	+
Maturity	+	+	+	+	+	+	~

Table A.1: *Summary of existing tools.*

and we exposed some of the most relevant design and implementation details. Finally, we compared our proposal with some other feature modelling tools and we concluded that this is the first tool integrating different solvers to optimize the analysis of feature models. Although FAMA is not a mature tool yet, its promising capabilities of extensibility, interoperability and integration make it a tool to take into account in the future.

Several challenges remain for our future work. The integration of new solvers and new analysis operations are currently in process. We are also studying licenses issues in order to release our tool. We really trust that formal semantics [86, 87] are needed for the verification of our framework and we are working on that direction too.

Appendix B
Mathematical notes

†1

B.1 Z Notation

The Z formal specification language is based on set theory and first-order predicate calculus [63]. It extends the use of these languages by allowing an additional mathematical type known as the schema type. Z schemas have two parts: the upper declarative part, which declares variables and their types, and the lower predicate part, which relates and constraints those variables. The type of any schema can be considered as the Cartesian product of its variables, without any notion of order, but constrained by the predicates. Modularity is facilitated in Z by allowing schemas to be included within other schemas. We can select a variable of a schema instance by writing *schemaInstance.var*.

To introduce a type in which we wish to abstract away from the actual elements of the type, we use the notion of a given set. For instance, we write $[VarName]$ to represent the set of all possible variable names. If we wish to state that a variable ranges over some finite set of values or an ordered pair of values we write $x : \mathbb{F}\, Feature$ and $x : Feature \times Feature$, respectively.

A summary of the notation used in this dissertation is presented in Table §B.1. For a more complete treatment of the Z language, the reader is referred to one of the numerous texts such as [107].

†1 This chapter is partially inspired by [4, 5]

Declarations	
$a : A$	Declarations
$A == B$	A is an alias for B
Logic	
$p \bigwedge q$	Logical conjunction
$p \bigvee q$	Logical disjunction
$\neg p$	Logical negation
$p \Rightarrow q$	Logical conditional
$p \Leftrightarrow q$	Logical biconditional
Quantification	
$\forall x : A \mid q(x) \bullet p(x)$	Universal quantification. Equivalent to the following: $\forall x \bullet x \in A \wedge q(x) \Rightarrow p(x)$
$\exists x : A \mid q(x) \bullet p(x)$	Existential quantification. Equivalent to the following: $\exists x \bullet x \in A \wedge q(x) \wedge p(x)$
$\exists_1 x : A \mid q(x) \bullet p(x)$	Unique existential quantification
Sets	
$x \in A$	Set membership
$\varnothing$	Empty set
$A \subseteq B$	Set inclusion
$\{x, y, \ldots\}$	Set of elements
(x, y)	Ordered pair
$A \times B$	Cartesian product
$\mathbb{P} A$	Powerset of A
$\mathbb{F} A$	Finite powerset of A
$\{ x : A \mid q(x) \bullet e(x)\}$	The set $\{ e(x) \mid x \in A \wedge q(x)\}$
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$\bigcap A$	Generalised or distributive intersection
$\bigcup A$	Generalised or distributive union
$\#A$	Cardinal of set A
Relations and Functions	
$A \rightarrow B$	Total function
$A \nrightarrow B$	Partial function
$\operatorname{dom} f$, $\operatorname{ran} f$	Domain and range of a function f

Table B.1: *Summary of the notation used in this dissertation.*

Appendix C
Acronyms

BDD. Binary Decision Diagram.

CNF. Conjunctive Normal Form.

CSP. Constraint Satisfaction Problem.

FAMA. FeAture Model Analyser

FM. Feature Model

EMF. Eclipse Modeling Framework.

SAT. Satisfiability Problem.

SPL. Software Product Line.

OMG. Object Management Group.

RACER. Renamed ABox and Concept Expression Reasoner.

UML. Unified Model Language.

XML. eXtensible Mark–up Language.

XMI. XML Metadata Interchange.

Bibliography

[1] *1st International Workshop on Visualisation in Software Product Line Engineering. In conjunction to Sofware Product Line Conference*, 2007.

[2] G. Aldekoa, S. Trujillo, G. Sagardui, and O. Díaz. Experience measuring maintenability in software product lines. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[3] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 201–210, New York, NY, USA, 2006. ACM Press.

[4] J. L. Arjona. *Bringing the knowledge on the web to sotware agents. A framework for developing semantic wrappers*. PhD thesis, Departament of Computer Languages ans Systems (University of Seville), March 2005.

[5] J. L. Arjona, R. Corchuelo, D. Ruiz, and M. Toro. From wrapping to knowledge. *IEEE Trans. Knowl. Data Eng.*, 19(2):310–323, 2007.

[6] T. Asikainen, T. Mannisto, and T. Soininen. A unified conceptual foundation for feature modelling. In *Software Product Line Conference, 2006 10th International Conference*, pages 31–40. IEEE Press, 2006.

[7] D. Avison, F. Lau, M. Myers, and P. Nielsen. Action research. *Communications of the ACM*, 42(1):94–97, January 1999.

[8] B. Barták. Theory and practice of constraint propagation. In *Proceedings of the 3rd Workshop on Constraint Programming for Decision and Control*, pages 7–14, 2001.

[9] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer–Verlag, 2005.

[10] D. Batory. A tutorial on feature oriented programming and the ahead tool suite. In *Summer school on Generative and Transformation Techniques in Software Engineering*, 2005.

[11] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12): 45–47, December 2006.

[12] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[13] D. Benavides, A. Durán, M.A. Serrano, and C. Montes-Oca. Quality of service variability in system families based on web services. In *Simposio de Informática y Telecomunicaciones (SIT)*, pages 205–218, Sevilla, Spain, 2002.

[14] D. Benavides, A. Ruiz-Cortés, and D. Batory. Automated analysis of software product lines using feature models: Applications, current solutions and challenges. *IEEE Computer (submitted)*, 2007.

[15] D. Benavides, A. Ruiz-Cortés, R. Corchuelo, and A. Durán. Seeking for extra-functional variability. In *Proceedings of the ECOOP Workshop on Modeling Variability for Object-Oriented Product Lines*, Darmstadt, Germany, 2003.

[16] D. Benavides, A. Ruiz-Cortés, R. Corchuelo, and O. Martín-Díaz. Spl needs an automatic holistic model for software reasoning with feature models. In *International Workshop on Requirements Reuse in System Family Engineering*, pages 27–33, Madrid, Spain, 2004. Universidad Politécnica de Madrid.

[17] D. Benavides, A. Ruiz-Cortés, and C. Muller. The triple schizophrenia of the software engineering researcher. In *1st Workshop on Philosophical Foundations of Information Systems Engineering (PHISE'05)*, June 2005.

[18] D. Benavides, A. Ruiz-Cortés, O. Martí n, and J. Bermejo. A firts approach to build product lines of MOWS. In *Proceedings of the ZOCO02 workshop*, El Escorial, Madrid, 2002.

[19] D. Benavides, A. Ruiz-Cortés, M.A. Serrano, and C. Montes de Oca. A first approach to build product lines of multi-organizational web based systems (MOWS). In *Innovative Internet Community Systems, IICS 2004*, volume 3473 of *Lecture Notes in Computer Sciences*, pages 91–98. Springer–Verlag, 2005.

[20] D. Benavides, A. Ruiz-Cortés, B. Smith, B. O'Sullivan, and P. Trinidad. Computational issues on the automated analysis of feature models using constraint programming. *In preparation*, 2007.

[21] D. Benavides, A. Ruiz-Cortés, and M. Toro. Aplicando la filosofía de las ciencias de la complejidad a la ingeniería del software. In *Primer Workshop en Métodos de Investigación y Fundamentos Filosóficos en Ingeniería del Software MIFISIS*, pages 97–106. Universidad Rey Juan Carlos, Spain, 2002.

[22] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Coping with automatic reasoning on software product lines. In *Proceedings of the 2nd Groningen Workshop on Software Variability Management*, November 2004.

[23] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, volume 3520 of *Lecture Notes in Computer Sciences*, pages 491–503. Springer–Verlag, 2005.

[24] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Using constraint programming to reason on feature models. In *The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE*, pages 677–682, 2005.

[25] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 367–376, 2006.

[26] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Proceedings of the Workshop on Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–45, 2006.

[27] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, chapter Using Java CSP Solvers in the Automated Analyses of Feature Models, pages 389–398. Springer–Verlag, 2006.

[28] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 129–134, 2007.

[29] D. Benavides, S. Trujillo, and P. Trinidad. On the modularization of feature models. In *Proceedings of the First European Workshop on Model Transformation. Software Product Line Conference*, September 2005.

[30] J. Benavides. El desarrollo local sostenible con calidad de vida. In *proceedings of the XII congreso iberoamericano de urbanismo*, 2006.

[31] J. Bermejo, P. Trinidad, D. Benavides, and A. Ruiz-Cortés. *Software product lines in action*, chapter Experience reports: Telvent. Springer, June 2007.

[32] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38 (4), 2006.

[33] J. Bowen and M. Hinchey. Ten commandments of formal methods... ten years later. *Computer*, 39(1):40–48, 2006.

[34] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[35] R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[36] F. Cao, B. Bryant, C. Burt, Z. Huang, R. Raje, A. Olson, and M. Auguston. Automating feature-oriented domain analysis. In *International Conference on Software Engineering Research and Practice (SERP'03)*, pages 944–949, June 2003.

[37] V. Cechticky, A.Pasetti, O. Rohlik, and W. Schaufelberger. XML-based feature modelling. In *Software Reuse: Methods, Techniques and Tools: 8th ICSR 2004. Proceedings*, volume 3107 of *Lecture Notes in Computer Sciences*, pages 101–114. Springer–Verlag, 2004.

[38] M. Cengarle, P. Graubmann, and S. Wagner. Semantics of UML 2.0 interactions with variabilities. *Electronic Notes in Theoretical Computer Science*, 160:141–155, 2006.

[39] G. Chastek, P. Donohoe, K.C. Kang, and S. Thiel. Product Line Analysis: A Practical Introduction. Technical Report CMU / SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, June 2001.

[40] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, August 2001.

[41] S. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[42] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, pages 156–172, 2002.

[43] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison–Wesley, may 2000. ISBN 0–201–30977–7.

[44] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Proceedings of the Third Software Product Line Conference 2004*, volume 3154 of *Lecture Notes in Computer Sciences*, pages 266–282. Springer–Verlag, 2004.

[45] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2), 2005.

[46] K. Czarnecki, S. Helsen, and U.W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[47] K. Czarnecki, C. Kim, and K. Kalleberg. Feature models are views on ontologies. In *proceedings of the 10th International Software Product Line Conference (SPLC)*, Baltimore, Maryland, USA, August 2006.

[48] K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA 2005*, 2005.

[49] A. van Deursen and P. Klint. Domain–specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10 (1):1–17, 2002.

[50] O. Díaz, S. Trujillo, and F.I. Anfurrutia. Supporting production strategies as refinements of the production process. In *Proceedings of the Sofware Product Line Conference (SPLC 2005)*, volume 3714 of *Lecture Notes in Computer Sciences*. Springer–Verlag, 2005.

[51] O. Díaz, S. Trujillo, and I. Azpeitia. User-Facing Web Service Development: A Case for a Product-Line Approach. In Boualem Benatallah and

Ming-Chien Shan, editors, *Technologies for E-Services, 4th VLDB International Workshop (VLDB-TES 2003)*, volume 2819 of *LNCS*, pages 66–77. Springer-Verlag, 2003.

[52] O. Diaz, S. Trujillo, and S. Perez. Turning portlets into services: The consumer profile. In *Proceedings of the 16th International World Wide Web Conference*, 2007.

[53] A. Durán, D. Benavides, and J. Bermejo. Applying system families concepts to requirements engineering process definition. In *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 140–151. Springer-Verlag, 2004.

[54] L. Etxeberria, G. Sagardui, and L. Belategi. Modelling variation in quality attributes. In *Proceedings of the First International Workshop on Variability Modelling of Software–intensive Systems (VAMOS)*, 2007.

[55] P. Fernandez and M. Resinas. James project. *Available at http://jamesproject.sourceforge.net/*, 2002-2005.

[56] E. C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, April 1997.

[57] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *Proceedings of the ACM SIGSOFT First Alloy Workshop*, pages 71–80, 2006.

[58] M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, Canada, 1998.

[59] J. Gu, P. Purdom, J. Franco, and B. Wah. *Satisfiability Problem: Theory and Applications*, chapter Algorithms for the Satisfiability (SAT) Problem: A Survey, pages 19–152. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[60] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[61] P. Van Hentenryck and V. Saraswat. Strategic directions in constraint programming. *ACM Comput. Surv*, 28(4):701–726, December 1996.

[62] P. Heymans, P.Y. Schobbens, J.C. Trigaux, R. Matulevicius, A. Classen, and Y. Bontemps. Towards the comparative evaluation of feature diagram languages. In *Proceedings of the Software and Services Variability Management Workshop - Concepts, Models and Tools*, 2007.

[63] ISO. Information technology – z formal specification notation – syntax, type system and semantics. iso / iec 13568:2002, 2002.

[64] S. Jarzabek, W. Ong, and H. Zhang. Handling variant requirements in domain modeling. *The Journal of Systems and Software*, 68(3):171–182, 2003.

[65] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU / SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[66] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature–oriented reuse method with domain–specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[67] K.C. Kang, J. Lee, and P. Donohoe. Feature–Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, July / August 2002.

[68] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, April 1993.

[69] S. Kotha. From mass production to mass customization: The case of the national industrial bicycle company of japan. *European Management Journal*, 14(5):442–450, 1996.

[70] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, July 2003.

[71] K. Lee, K.C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. *Lecture Notes in Computer Science*, 2319:62–77, 2002.

[72] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[73] M. Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, volume 2379 of *Lecture Notes in Computer Sciences*, pages 176–187, San Diego, CA, 2002. Springer–Verlag.

[74] M. Mannion and J. Camara. Theorem Proving for Product Line Model Verification. In *Proceedings of the Third Software Product Line Conference (SPLC03)*, volume 3014 of *Lecture Notes In Computer Sciences*, pages 211–224. Springer–Verlag, 2003.

[75] D. Mcilroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[76] A. Metzger, P. Heymans, K. Pohl, and P. Y. Schobbens. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (to appear)*, 2007.

[77] V. Pankratius. *Software product lines for digital information products*. Karlsruhe University Press, 2007.

[78] J. Pena, M. Hinchey, and A. Ruiz-Cortés. Multiagent system product lines: Challenges and benefits. *Communications of the ACM*, 49(12):82–84, December 2006.

[79] J. Pena, M. Hinchey, A. Ruiz-Cortés, and P. Trinidad. Building the core architecture of a multiagent system product line: With an example from a future nasa mission. In *7th International Workshop on Agent Oriented Software Engineering*, Lecture Notes in Computer Sciences. Springer–Verlag, 2006.

[80] G. Pesant. Counting solutions of CSPs: A structural approach. In *Proceedings og the International Joint Conferences on Artificial Intelligence*, pages 260–265, 2005.

[81] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer–Verlag, 2005.

[82] C. Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.

[83] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with UML multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, June 2002.

[84] M. Riebisch, D. Streitferft, and I. Pashov. Modeling variability for object-oriented product lines. In *ECOOP 2003 Workshop Reader*, volume 3013 of *Lecture Notes in Computer Sciences*. Springer–Verlag, 2004.

[85] A. Ruiz-Cortés. *Una Aproximación Semicualitativa al Tratamiento Automático de Requisitos de Calidad*. Phd thesis, Universidad de Sevilla, 2002.

[86] P. Schobbens, P. Heymans, and J. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, 2006.

[87] P. Schobbens, P. Heymans, J.C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb 2007.

[88] D. Streitferdt, M. Riebisch, and I. Philippow. Details of formalized relations in feature models using OCL. In *Proceedings of 10th IEEE International Conference on Engineering of Computer–Based Systems (ECBS 2003), Huntsville, USA. IEEE Computer Society*, pages 45–54, 2003.

[89] J. Sun, H. Zhang, Y.F. Li, and H. Wang. Formal semantics and verification for feature modeling. In *Proceedings of the ICECSS05*, 2005.

[90] J. Trigaux, P. Heymans, P. Schobbens, and A. Classen. Comparative semantics of feature diagrams: FFD vs. vDFD. In *Proceedings of the Fourth International Workshop on Comparative Evolution in Requirements Engineering (CERE)*, 2006.

[91] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Agile debugging of feature models. *Journal of System and Software (conditionally accepted)*, 2007.

[92] P. Trinidad, D. Benavides, and A. Ruiz-Cortés. Improving decision making in software product lines product plan management. In J. Dolado, I. Ramos, and J. Cuadrado-Gallego, editors, *Proceedings of the V ADIS 2004 Workshop on Decision Support in Software Engineering*, volume 120. CEUR Workshop Proceedings (CEUR-WS.org), 2004.

[93] P. Trinidad, D. Benavides, and A. Ruiz-Cortés. A first step detecting inconsistencies in feature models. In *CAiSE Short Paper Proceedings*, 2006.

[94] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and M. Toro. Explanations for agile feature models. In X. Franch and K. Cooper, editors, *1st International Workshop on Agile Product Line Engineering (APLE'06)*, 2006.

[95] S. Trujillo. *Feature Oriented Model Driven Product Lines*. Doctor of philosophy, University of the Basque Country, San Sebastián, Spain, 2007.

[96] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *Generative Programming and Component Engineering (GPCE).*, pages 191–200, New York, NY, USA, 2006. ACM Press.

[97] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.

[98] M. Tseng and J. Jiao. *Handbook of Industrial Engineering: Technology and Operations Management*, chapter Mass Customization, page 685. Wiley, 2001.

[99] F. van der Linden. Software product families in Europe: The Esaps & Café Projects. *IEEE Software*, 19(4):41–49, 2002.

[100] T. von der Massen and H. Lichter. Requiline: A requirements engineering tool for software product lines. In F. van der Linden, editor, *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, volume 3014 of *Lecture Notes in Computer Sciences*, Siena, Italy, 2003. Springer–Verlag.

[101] T. von der Massen and H. Lichter. Deficiencies in feature models. In Tomi Mannisto and Jan Bosch, editors, *Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004.

[102] T. von der Massen and H. Litcher. Determining the variation degree of feature models. In *Software Product Lines Conference*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 82–88, 2005.

[103] D. Wagelaar. Towards context-aware feature modelling using ontologies. In *Proceedings of the MoDELS 2005 International workshop on MDD for Software Product Lines*, 2005.

[104] D. Wagelaar and V. Jonckers. Explicit platform models for MDA. In *MODELS 2005 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005.

[105] H. Wang, Y. Li, J. Sun, H. Zhang, and J. Pan. A semantic web approach to feature modeling and verification. In *Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, November 2005.

[106] J. White, D.C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating product-line variant selection for mobile devices. In *Proceedings of the 11th Annual Software Product Line Conference (to appear)*, 2007.

[107] J. Wookcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice–Hal, 1996.

[108] W. Zhang, H. Mei, and H. Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*, 11(3):205–220, June 2006.

[109] W. Zhang, H. Zhao, and H. Mei. A propositional logic-based method for verification of feature models. In J. Davies, editor, *ICFEM 2004*, volume 3308, pages 115–130. Springer–Verlag, 2004.

[110] W. Zhao, B. Bryant, F. Cao, R. Raje, M. Auguston, C. Burt, and A. Olson. Grammatically interpreting feature compositions. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)*, 2004.

Index

This document was typeset on 2007/5/27 using RC–BOOK α2.10 for LaTeX$_{2\epsilon}$.
Should you want to use this document class, please send mail to
`contact@tdg-seville.info`.