Functional and Performance Testing of Feature Model Analysis Tools

Extending the FaMa Ecosystem

Sergio Segura Rueda

Universidad de Sevilla

European Doctoral Dissertation
Advised by
Dr. David Benavides Cuevas
and
Dr. Antonio Ruiz Cortés

First published in December 2010 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.2.5 [Software Engineering]: Testing and Debugging;

General Terms: Algorithms, Design, Performance, Verification.

Additional Key Words and Phrases: software product lines, feature models, automated analysis, software testing, test data generation.

Support: This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and SETI (TIN2009-07366) and by the Andalusian Government under project ISABEL (TIC-2533).

Don David Benavides Cuevas y Don Antonio Ruiz Cortés, profesores Titulares del Área de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla,

HACEN CONSTAR

que Don Sergio Segura Rueda, Ingeniero en Informática por la Universidad de Sevilla, ha realizado bajo nuestra supervisión el trabajo de investigación titulado

Functional and Performance Testing of Feature Model Analysis Tools. Extending the FaMa Ecosystem

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

Fdo. Dr. David Benavides Cuevas y Dr. Antonio Ruiz Cortés
Área de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
Sevilla, Diciembre de 2010

Yo, Sergio Segura Rueda, con DNI número 28.807.980-C,

DECLARO

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

Functional and Performance Testing of Feature Model Analysis Tools. Extending the FaMa Ecosystem

Lo cual firmo en Sevilla, Diciembre de 2010.

Fdo. Sergio Segura Rueda

De acuerdo a la normativa de la Universidad de Sevilla y el Departamento de Lenguajes y Sistemas Informáticos, el presente documento de tesis cumple los requisitos para la modalidad de *compendio de publicaciones relevantes* y *mención europea*. No obstante, la redacción final presentada sigue un esquema tradicional para una mejor comprensión de las contribuciones del candidato.

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. Arnaud Gotlieb (INRIA, France)
- Dr. Robert M. Hierons (University of Brunel, UK)

UNIVERSIDAD DE SEVILLA

The committee in charge of evaluating the dissertation presented by Sergio Segura Rueda 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 ____________________________________.

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

To my family
To Isabel

Contents

List of Figures

List of Tables

Acknowledgements

I have expected this moment for a long time. The sweet moment in which, with the feeling of the work done, I could spend some minutes thanking those people that I really care about, those that have made this thesis possible and to whom I owe my deepest gratitude.

First and foremost I am heartily thankful to my supervisors, Dr. David Benavides and Dr. Antonio Ruiz. They were the first to identify my research skills and the ones who gave me the opportunity to develop them. This thesis is undoubtedly the result of their encouragement, guidance and support.

During the research period, I have also benefited from being surrounded by a group of great people that have contributed, in one way or another, to make this thesis possible. Pablo Trinidad, Jose M. García, Carlos Müller, José A. Parejo, José A. Galindo or Jesus Galán are some of the fellow students with whom I spent more time and I would like thank them for their patient and support. Also, I wish to thank the rest of the members of the ISA group for their companionship and great help along this time. I also want to show my gratitude to the other members of the Department of Computer Language and Systems and the members of the thesis committee for their cheerful willingness to evaluate and improve my work.

This thesis has also fed off the insights and directions of colleagues from other universities and research centres. I must express special gratitude to the professor Robert M. Hierons who welcomed me warmly in his lab during my research stay in London. This thesis is also the result of his experience, sound advice and good ideas.

Finally, and more importantly, I must thank my family and friends for being always there whenever I needed them. Thanks mum, for instilling in me the hunger for knowledge and fighting so hard for giving me the opportunity that you never had. Thanks Isabel, for being the most beautiful woman in the world and loving me so much.

Abstract

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

Popular saying,

Software product line engineering is an approach to develop families of related software products by reusing a common set of features instead of building each product from scratch. Feature models play a key role in this paradigm by providing a high-level representation of all the products of the product line. The automated extraction of information from feature models is a thriving topic that has called the attention of researchers for the last twenty years. During this time, numerous operations, techniques and tools to get information from feature models have proliferated and a whole community has been built around what has been called the automated analysis of feature models. Currently, the rapid progress of this discipline is naturally leading to an increasing concern about the quality of feature model analysis tools. The goal now is not simply developing basic research prototypes but solid and high quality analysis tools in terms of absence of bugs and performance. In this context, current testing methods are mainly random and guided by intuition rather than by well-studied testing techniques. This makes testing conclusions rarely rigorous and verifiable weakening the value and scope of research contributions and reducing the user's confidence in the correctness of analysis tools.

The main goal of this thesis is to join knowledge from the software testing and feature modelling communities into a set of contributions that lead the automated analysis of feature models to a new level of maturity. To that purpose, we present a set of techniques, algorithms and tools to support functional and performance testing of feature model analysis tools. These contributions are the results of the application of several classical and innovative testing techniques to the context of the analysis of feature models. Regarding functional testing, we present a test suite and automated test data generator enabling the efficient detection of faults in analysis tools. Regarding performance testing, we present an evolutionary algorithm for the generation of computationally-hard feature models to reveal the deficiencies of tools in pessimistic cases. These contributions have been evaluated using extensive and rigorous experiments that reveal the efficacy and efficiency of our approach. Among other results, we detected two faults in FaMa and another two in SPLOT, two popular analysis tools widely used in the community of automated analysis of feature models. Our contributions have been integrated into a framework, BeTTy, built as a part of the FaMa ecosystem, a tool suite for the analysis of feature models developed in the bosom of our research group.

Resumen

Sólo tendrás una oportunidad
de dar una primera impresión.

Dicho popular,

La ingeniería de líneas de productos es un paradigma de desarrollo orientado a construir familias de sistemas software que reutilicen características comunes en lugar de construir cada producto desde cero. Un aspecto fundamental para la gestión de una línea de productos es utilizar un modelo que permita representar todos los posibles productos que pueden derivarse de ella. Uno de los modelos más populares usados para tal fin son los denominados modelos de características. La información extraída de estos modelos es utilizada para tomar decisiones importantes a lo largo del desarrollo y gestión de una línea de productos software.

La extracción automática de información de modelos de características es un tema de investigación activo que ha atraído la atención de numerosos investigadores en los últimos veinte años. Durante este tiempo, se han presentado un gran número de operaciones, técnicas y herramientas y se ha consolidado toda una comunidad alrededor de lo que se ha denominado análisis automático de modelos de características. Actualmente, los avances en éste área están llevando a una mayor preocupación por la calidad de las herramientas de análisis. Los prototipos de investigación ya no son suficiente y ahora se buscan solidas herramientas de análisis en términos de ausencia de errores y rendimiento. Sin embargo, las pruebas software empleadas en este campo son fundamentalmente aleatorias y guiadas por la intuición de los propios desarrolladores más que por técnicas maduras para el diseño de datos de prueba. Esto hace que las conclusiones de las pruebas sean difícilmente rigurosas y verificables a la vez que limitan el alcance y el valor de las contribuciones científicas en el análisis de modelos de características. De igual forma, la arbitrariedad u omisión de las pruebas en este campo reducen notablemente la confianza de los usuarios sobre el correcto funcionamiento de las herramientas de análisis.

El objetivo de esta tesis es combinar el conocimiento de las comunidades de pruebas software y modelado de características en una serie de contribuciones que permitan llevar el análisis automático de modelos de características a un nuevo nivel de madurez. Con este fin, en esta memoria presentamos una serie de técnicas, algoritmos y herramientas para la realización de pruebas funcionales y de rendimiento en herramientas de análisis de modelos de características. Estas contribuciones son el resultado de la aplicación de varias técnicas clásicas e innovadoras de pruebas software en el contexto de análisis de modelos de características. Para facilitar las pruebas funcionales, presentamos un conjunto de casos de prueba así como un generador automático de datos de prueba para la detección de errores en las herramientas de análisis. En lo que se refiere a pruebas de rendimiento, presentamos un algoritmo evolutivo para la generación de modelos de características difíciles de procesar en términos de tiempo y memoria requeridos para su análisis. Etas herramientas han sido evaluadas experimentalmente de forma rigurosa demostrando la viabilidad de la propuesta. Entre otros resultados, nuestras herramientas nos han permitido detectar dos errores en FaMa y otros dos en SPLOT, dos herramientas para el análisis de modelos de características ampliamente usadas por la comunidad. Nuestras contribuciones han sido integradas en un framework, BeTTy, desarrollado como parte del ecosistema FaMa, un conjunto de herramientas para el análisis de modelos de características desarrolladas por nuestro grupo de investigación.

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

In this dissertation, we report on our work to develop a set of techniques, algorithms and tools to support functional and performance testing of feature model analysis tools. In this chapter, we first introduce the topics that constitute the context of our research work in Section §1.1. In Section §1.2, we summarize our main contributions and publications. Finally, we describe the structure of the dissertation in Section §1.3.

1.1 Research context

In the next sections, we briefly present the main concepts that we will use throughout the rest of this dissertation. In Section §1.1.1, we present software product lines. Feature models and their analyses are described in Section §1.1.2. Finally, we introduce the main concepts concerning testing on the analysis of feature models in Section §1.1.3.

1.1.1 Software product lines

Economic and social progress has brought significant changes to the ways in which goods are produced. Centuries ago, craftsmen were the responsible for the building of each product from the beginning to the end. This means that they had to know everything about the assembly process and the creation of the individual parts that composed each product. With the industrial revolution, a new production paradigm emerged, so-called mass production. *Mass production* can be defined as the creation of large amount of the same product using standardized processes and techniques (i.e. assembly lines) intended to reduce costs and time to market. Examples of mass production can be found in almost all commercial areas such as transport, communications or food (see Figure §1.1). In a globalized world, however, the highly competitive and segment–oriented market is starting to show the deficiencies of mass production when it comes to respond to the demands for variety and customization of customers. To address these needs, industry is currently involved in a shift from mass production to a more flexible paradigm referred to as *mass customization*. Tseng and Jiao define mass customization as "*producing goods and services to meet individual customers' needs with near mass production efficiency*" [174]. Figure §1.2 depicts an example of how mass customization is offered to the consumers of mobile phones by means of web configurators in which users can select the features that better meet their needs. The story of software product lines is about the application of mass customization to the production of software [15, 21].

Figure 1.1: *Some pictures illustrating mass production.*

Software product lines (a.k.a. software product families) has emerged as one of the most promising reuse-based software development paradigm for a major improvement of the productivity of IT industries, enabling them to handle the diversity of global market and reduce the development costs and the amount of time to market [45, 162]. In particular, software product lines are based on two key ideas. First, most software systems are not new. In fact, software products usually share a number of features. Consider, for instance, the common features (a.k.a.

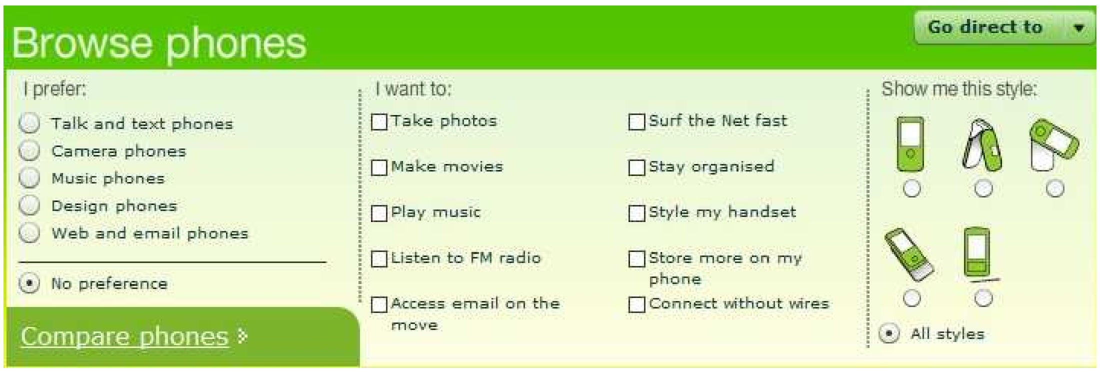


Figure 1.2: *Example illustrating mass customization in the mobile phone industry.*

commonalities) that can be found in current on-line purchase systems (e.g. catalogue management, credit card validation, data access, etc.). Second, many companies are becoming market segment–oriented. In this context, different variants of the same products are launched to the market by combining different features in response to different customers' preferences. As an example, consider the software loaded in mobile phones in which hundreds of different models are built by combining a common set of reusable features: calls, messaging, Bluetooth, MP3, 3G, games, etc. Based on these ideas, software product line engineering focuses on the systematic development of families of products rather than producing each product one by one from scratch [45, 129]. To this aim, product variants are built using reusable assets which usually include frameworks and components. More specifically, Clements and Northrop [45] define a software product line as follows:

"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

Software product lines are traditionally developed and managed within the boundaries of an organization. However, the success of a software product line may lead a company to share their core assets (usually referred to as *platform*) with the community to promote external contributions to the product line. When this occurs, companies transition from a software product line approach to a so-called *software ecosystem* approach [30, 111]. Among other benefits, software ecosystems increase the value and attractiveness of products for new users and accelerate innovation. Software ecosystems have emerged in different areas like those of operating systems (e.g. iPhone OS) or Web applications (e.g. Google AppEngine). However, most software ecosystems are derived from classical desktop applications that achieves success in the market and are then opened up to promote contributions from their community of users (e.g. MS Office Suite). The adoption of a software ecosystem implies benefits but also new management challenges like the coordination of internal and external developers and the collaboration with partner companies and independent solution vendors.

1.1.2 Automated analysis of feature models

The products of a software product line can be specified in terms of *features*. A feature is defined as an increment in product functionality [11]. For instance, the software system of a mobile phone could be defined by the set of features that it supports as follows: {Calls, Messaging, MP3}. This could intuitively means that the software product provides support for calls, messaging and playing MP3 respectively.

Key to software product lines is to capture commonalities (i.e. common features) and variabilities (i.e. variant features) of the systems that belong to the product line. To this aim, *feature models* [85] are used. A *feature model* is a compact representation of all the products of a software product line in terms of features. Figure §1.3 depicts a simplified example feature model inspired by the mobile phone industry. The model illustrates how features are used to specify and build software for mobile phones. The software loaded in the phone is determined by the features that it supports. As illustrated, a feature model consists of:

- Nodes representing product features.
- Relationships between a parent feature and its child features.
- Cross-tree constraints that are typically inclusion or exclusion statements of the form : "If feature A is included, then feature B must also be included (or excluded)".

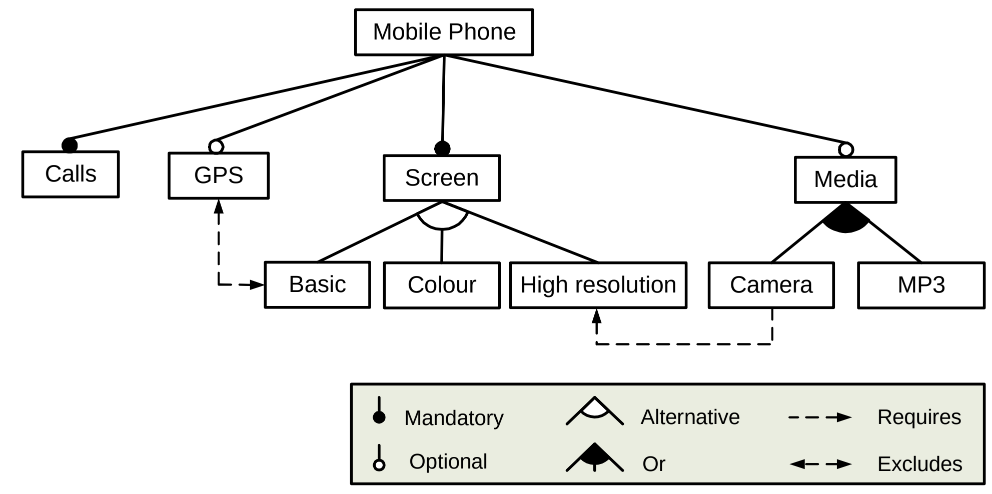


Figure 1.3: *Mobile phone feature model.*

Feature models were first introduced in 1990 as a part of the FODA (Feature–Oriented Domain Analysis) method [85]. Since then, feature modelling has been widely adopted by the software product line community and a number of extensions have been proposed in attempts to improve properties such as succinctness and naturalness [140]. At the time of writing this dissertation, two main groups of feature model notations are used in the literature: basic feature models [71] and cardinality-based feature models [50]. As a possible complement to both of them, a third notation, *extended feature models* [18], proposes adding extra functional information to the models by means of feature attributes.

Right from the introduction of feature models by Kang back in 1990, the manual manipulation of these was recognized as a difficult and error-prone task [85]. Since then, a number of approaches for the automated analysis of feature models have proliferated. The automated analysis of feature models deals with the computer–aided extraction of information from feature models. From the information obtained, marketing, managerial and technical decisions are derived all along the software product line lifecycle [15]. Typical operations of analysis allow us to know whether a feature model is void (i.e. it represents no products), what is the number of products represented by a feature model or whether a model contains any errors. Recent surveys have identified up to 30 different analysis operations on feature models proposed in the literature [21]. Common analysis techniques are those based on propositional logic, constraint programming or description logic [21]. The applications supporting these analysis capabilities are usually referred to as *feature model analysis tools*. There exists a variety of commercial and open source feature model analysis tools such as *AHEAD Tool Suite* [4], *Big Lever Software Gears* [27], *FaMa Framework* [58], *Feature Model Plug-in* [61], *pure::variants* [131] and SPLOT [159].

1.1.3 Testing of feature model analysis tools

We human beings are by nature not perfect and so is not the software that we develop. Even the most experienced programmer can make mistakes and introduce faults in his programs [14, 47, 117]. The process of evaluating a program with the intent of finding faults is referred to as *software testing* [117]. Software testing plays a key role in the development of software to evaluate whether the program meets its requirements, both functional and non-functional, and to gain confidence about the absence of bugs in the program. In fact, it is acknowledged that software testing consumes more than 50% of the total development time and budget of a software project [14, 117]. Despite this massive investment, however, we must admit that it is still impractical, often impossible, to detect all the faults of a program [14, 47, 117].

Software testing is performed by means of test cases. A *test case* is a set of test inputs and expected outputs developed to verify compliance with a specific requirement [47, 117]. Test cases may be grouped into so-called *test suites*. A variety of testing techniques has been reported to assist on the design of effective test cases, i.e. those that will find more faults with less effort and time [14, 47, 117]. These techniques can be classified according to multiple factors such as knowledge of the source code (black–box, white–box, gray–box), source of information used (program–based, specification–based, interface–based), testing level (unit–level, integration–level, system–level) or degree of automation (manual vs. automated).

In this thesis, we will focus on the testing of feature model analysis tools at two levels, namely:

Functional testing. During functional testing, analysis tools are evaluated to check whether the implemented analyses are actually performing the expected computation. Gaining confidence in the absence of faults in these tools is especially relevant since the information extracted from feature models is used all along the development process to support important decisions [12]. The main obstacles at this level are the lack of representative test data and the difficulty to check whether the output of an analysis is correct, so–called *oracle problem* [37, 189].

Performance testing. During performance testing, analysis tools are exercised to check how well they behave when dealing with different types of problems, e.g. execution time. From the results obtained, the strengths and weaknesses of the applications are highlighted helping researchers to improve their solutions and identify new research directions. One of the hardest challenges in this scenario is to find motivating input feature models that show the performance of tools in extreme situations, e.g. those maximizing the execution time or the memory consumption of the tools.

1.2 Contributions

In this section, we summarize the main contributions of our research work. These contributions have been published in relevant journals, conferences and workshops.

1.2.1 Summary of contributions

This thesis continues the work on the analysis of feature models carried out by our research group in the last years. This work has focused on the development of a product line of tools for the analysis of variability models and a framework, FaMa (FeAture Model Analyser), supporting it. Recently, however, we have realized of the benefits of opening up the FaMa product line to external contributors sharing the costs and benefits of innovation. Hence, we are currently involved in a shift from a product line strategy to a software ecosystem approach [30]. Figure §1.4 depicts a graphical representation of the FaMa ecosystem. The FaMa framework represents the core (also referred to as *platform* in the literature) of the ecosystem. Around it, a number of extensions are built. These can be mainly divided into analysis components and 3rd party tool integration usually carried out by external collaborators. At the time of writing this dissertation, FaMa is integrated into the visual editor MOSKitt feature modeller [113] and it is being integrated into the commercial tool pure::variants [131][†1]. One of the results of this dissertation is the integration of our contributions in the fields of functional and performance testing of feature model analysis tools into the FaMa ecosystem. In particular, we have endowed the ecosystem with a test suite, FaMa TeS (FaMa Test Suite), and a framework, *BeTTy (Benchmarking and TesTing on the analysis of feature models)* . Both resources have been released under open source license and are accessible from the BeTTy [25] and FaMa [58] Web sites.

The main goal of this dissertation is to provide a set of techniques, algorithms and tools to support the functional and performance testing of feature model analysis tools. In the pursuit of this goal, we have made the following contributions:

i. **FUNCTIONAL TESTING OF FEATURE MODEL ANALYSIS TOOLS**

   Problem statement: The lack of representative test data hinders the testing of feature model analysis tools.

   Contribution: We have developed a set of implementation–independent test cases for the detection of faults in feature model analysis tools. The main results of this contribution have been presented in the 5th Software Product Lines Testing Workshop [143] and the IET Software journal [150].

[†1]In the context of the DiVA European project (http://www.ict-diva.eu/)

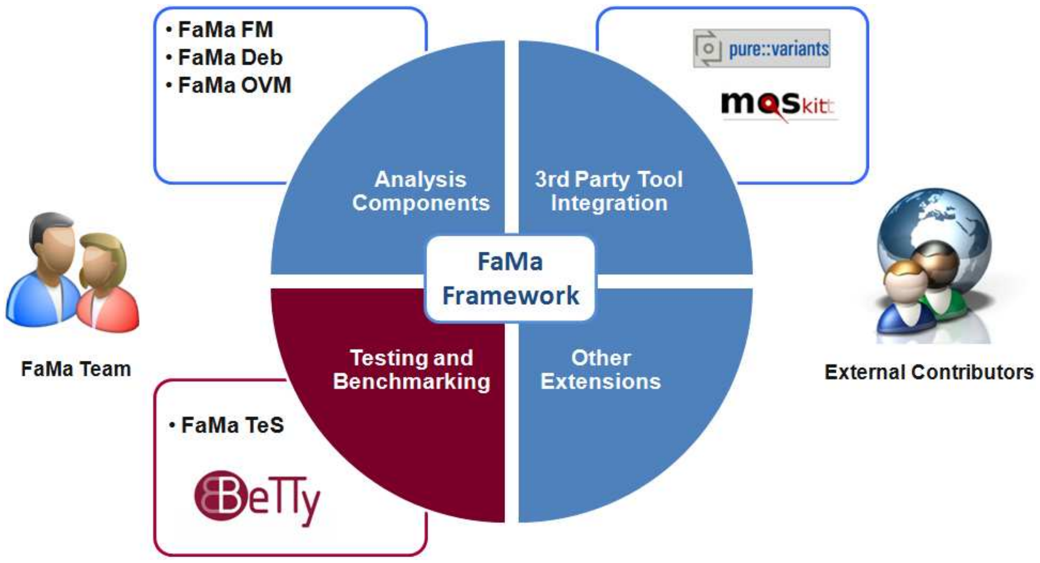


Figure 1.4: *The FaMa Ecosystem.*

Problem statement: The testing of feature model analysis tools is a hard and time–consuming task.

Contribution: We have developed an automated test data generator for the analyses of feature models and we have evaluated it using both artificial and real faults. The main results of this contribution have been presented in the Third International Conference on Software Testing, Verification and Validation [151] and in the Information and Software Technology journal [152, 153].

ii. **PERFORMANCE TESTING OF FEATURE MODEL ANALYSIS TOOLS**

Problem statement: There are not available hard feature models to evaluate the performance of analysis solutions.

Contribution: We have developed a novel evolutionary algorithm for the generation of computationally–hard feature models, e.g. those producing longest execution times or highest memory consumption. This allows users and developers to know the behaviour of tools in extreme situation revealing their real power. The mains results of these contribution are about to be submitted to a journal.

In addition to the aforementioned approaches, during our research period we have made some other contributions that are not included in this dissertation due to space constraints, namely:

i. **SURVEY AND TOOL SUPPORT ON THE ANALYSIS OF FEATURE MODELS**

Problem statement: The publications on the analysis of feature models are numerous and scattered hindering the progress of the discipline.

Contribution: We performed an exhaustive literature review and research agenda of the analysis of feature models 20 years after of their introduction. The main result of this contribution were presented in the XI Jornadas de Ingeniería del Software y Bases de Datos [20] and the Information Systems journal [21].

Problem statement: The analysis of feature models requires efficient tool support.

Contribution: We have worked actively in the development of techniques and tools for the analysis of feature models. The main results of this contributions have been published in several international workshops and book chapters [22–24, 145, 169] as well as in the tool tracks of the XII Jornadas de Ingeniería del Software y Bases de Datos [170] and the 12th International Software Product Line Conference [168].

ii. **BENCHMARKING AND PERFORMANCE OPTIMIZATION**

Problem statement: The lack of standard benchmarks to compare the performance of different analysis solutions is hindering the progress of the community.

Contribution: We suggested the creation of a benchmark for the automated analyses of feature models and proposed a preliminary research agenda setting milestones and clarifying the types of contributions expected from the community. The main result of this contribution was presented in the Third International Workshop on Variability Modelling of Software-intensive Systems [154].

Problem statement: The analysis of feature models is a complex task.

Contribution: We presented an algorithm to reduce the size of the feature model prior to their analysis reducing the complexity of the analysis process. We also showed the gains in efficiency obtained when using this technique with different analysis tools. The main result of this contribution was presented in the First Workshop on Analyses of Software Product Lines [142].

iii. **OTHER AUTOMATED TREATMENTS AND VARIABILITY DOMAINS**

Problem statement: The merging of feature model requires automated support.

Contribution: We proposed using model transformation to automate the merging of feature models and presented a prototype implementation using graph transformations. The results of this contribution were presented in a LNCS book chapter [148] and the VII Jornadas sobre Programación y Lenguajes [147].

Problem statement: The synergies of service oriented applications and software product lines are still to be explored.

Contribution: We presented a taxonomy of variability points in Web Service Flows. This contribution was presented in the First International Workshop on Service Oriented Architectures and Product Lines [146].

1.2.2 Publications in chronological order

We next present a complete list of the publications derived from our research work in chronological order.

[2006]. During our first year of work, we focused on the automated support for the analysis of feature models. We proposed new analysis operations and compared the performance of different analysis techniques [22, 23, 169]. Also, we published a preliminary version of a survey on the analysis of feature models [20]. These works gave us the first hints about the lack of standard mechanisms to test and compare the performance of different analysis tools.

- **APLE'06**. P. Trinidad , D. Benavides, A. Ruiz-Cortes, *S. Segura* and M. Toro. Explanations for agile feature models. *1st International Workshop on Agile Product Line Engineering (APLE'06)*. Baltimore, Maryland, USA, 2006.
- **GTTSE'06**. D. Benavides, *S. Segura*, P. Trinidad and A. Ruiz-Cortés. Using Java CSP solvers in the automated analyses of feature models. *Post-proceedings Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*. LNCS, 4143:389- 398. Braga, Portugal, 2006.
- **SPLC'06**. D. Benavides, *S. Segura*, P. Trinidad and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. *Managing Variability for Software Product Lines: Working with Variability Mechanisms*, pages 39-45. Baltimore, Maryland, USA, 2006.
- **JISBD'06**. D. Benavides, P. Trinidad, A. Ruiz-Cortés and *S. Segura*. A survey on the automated analyses of feature models. *Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06)*, pages 367-376. Sitges, Barcelona (Spain), 2006.

[2007]. During this year, we released the first version of the analysis framework FaMa and presented it to the community in several publications [24, 145, 170]. During the development of the tool, we learned how difficult it was to gain confidence about the absence of errors in the implementation of analysis operations. This was the problem that motivated big part of this PhD dissertation. In the same year, we made some exploratory works. In particular, we explored the connections between web services and software product lines [146]. Also, we studied how model transformations could help to provide support for other automated treatments on feature models like model refactorings [144, 147].

- **VaMoS'07**. D. Benavides, *S. Segura*, P. Trinidad and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. *First International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'07),*pages 129-134. Limerick, Ireland, 2007.
- **OSSPL'07**. *S. Segura*, D. Benavides, A. Ruiz-Cortés and P. Trinidad. Open Source Tools for Software Product Line Development .*Open Source and Product Lines (OSSPL'07)*. Kyoto, Japan, 2007.
- **SOAPL'07**. *S. Segura*, D. Benavides, A. Ruiz-Cortés and P. Trinidad. A Taxonomy of Variability in Web Service Flows. *Service Oriented Architectures and Product Lines (SOAPL'07)*. Kyoto, Japan, 2007.

- **PROLE'07**. *S. Segura*, D. Benavides, A. Ruiz-Cortés and P Trinidad. Toward Automated Refactoring of Feature Models using Graph Transformations. *VII Jornadas sobre Programación y Lenguajes (PROLE'07)*, pages 275-284. Zaragoza, Spain, 2007.

- **JISBD'07**. P. Trinidad, D. Benavides, *S. Segura*, and A. Ruiz-Cortés. Fama: hacia el análisis automático de modelos de características. *In Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos.* Zaragoza, Spain, 2007.

- **DSDM'07**. *S. Segura*, D. Benavides, A. Ruiz-Cortés and M.J. Escalona. From Requirements to Web System Design. An Automated Approach using Graph Transformations. *Desarrollo de Software Dirigido por Modelos. 4^a Edición (DSDM'07)*, pages 61-69. Zaragoza, Spain, 2007.

[2008]. In this year, we presented our first contributions in the context of functional testing and performance on the analyses of feature models. In particular, we proposed the design of a test suite for the analysis of feature models and studied which testing techniques were more appropriate for that end [143]. Also, we proposed and algorithm to reduce the size of feature model prior to their analysis gaining efficiency [142]. We also presented FaMa in the Software Product Line Conference (SPLC), main forum of our community [168]. Furthermore, we proposed using graph transformation to automate the merging of feature models. This contribution was the result of our participation in the Second Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'07). After the event, the organizers competitively selected 4 papers out of the submissions of the participants to be included in a chapter of a special volume of Springer-Verlag Lecture Notes in Computer Science were our paper was published [148]. In the same year, we collaborated with Trinidad in proposing a tree-dimensional visualization of feature models [171]. We also collaborated with Ross-Frantz studying how to apply our knowledge about feature models to the analysis of Orthogonal Variability Models [136].

- **GTTSE'08**. *S. Segura*, D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated merging of feature models using graph transformations. *Postproceedings of the Second Summer School on Generative and Transformational Techniques in Software Engineering.* LNCS, 5235:489-505. Braga, Portugal, 2008.

- **SPLiT'08**. *S. Segura*, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools. a first step. *Proceedings of the Fifth International Workshop on Software Product Lines Testing (SPLiT'08)*, pages 36-39. Limerick, Ireland, 2008.

- **ASPL'08-1**. *S. Segura*. Automated analysis of feature models using atomic sets. *In proceedings of the First Workshop on Analyses of Software Product Lines (ASPL'08)*, pages 201-207. Limerick, Ireland, 2008.

- **ASPL'08-2**. F. Roos-Frantz and *S. Segura*. Automated Analysis of Orthogonal Variability Models. A First Step. *In proceedings of the First Workshop on Analyses of Software Product Lines (ASPL'08)*, pages 243-248. Limerick, Ireland, 2008.

- **SPLC'08**. P. Trinidad, D. Benavides, A. Ruiz-Cortés, *S. Segura*, and A. Jimenez. Fama framework. *In 12th Software Product Lines Conference - Tool track.* Limerick, Ireland, 2008.

- **ViSPLE'08**. P. Trinidad, A. Ruiz-Cortés, D. Benavides and *S. Segura*. Three-Dimensional Feature Diagrams Visualization. *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008).* Limerick, Ireland, 2008.

[2009]. We presented a roadmap for the development of a benchmark to compare the performance of different analysis solutions [154]. In this work, we identified challenges, milestones and types of contributions expected from the community. During this year, we also spent three months in a research stay at England and submitted several journal and conference papers that would be accepted the following year.

- **VaMoS'09**. *S. Segura* and A. Ruiz-Cortés. Benchmarking on the automated analyses of feature models: A preliminary roadmap. *In Third International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09)*, pages 137-143. Seville, Spain, 2009.

[2010]. This was the most prolific year in terms of publications. First, we presented a literature review on the analysis of feature models in the Information Systems journal [21]. Later, we presented a test suite for functional testing of feature model analysis tools in the IET Software journal [150]. We also got a paper accepted in the International Conference on Software Testing, Verification and Validation (ICST) [151]. This was a join work with the Professor Robert M. Hierons who supervised our work during our research stay in his lab in England. Our paper in ICST was extended in two directions. First, we reported on our experience using mutation testing for the evaluation of our test data generator in the Information and Software Technology Special Issue on Mutation Testing [152]. Second, we presented an improved version of our test data generator and compared it with our manual test suite in an article published in the Information and Software Technology journal [153]. In addition to this, we developed an evolutionary algorithm for the generation of hard problems to be used in performance evaluations. The results of this work are about to be submitted to a journal. Finally, we collaborated with Galindo in the extraction of hard and realistic feature models from Debian package repositories [63].

- **ICST'10**. *S. Segura*, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. *In International Conference on Software Testing, Verification and Validation*, pages 35-44. Paris, France, 2010. [Acceptation rate: 50/189 (26.5%)]

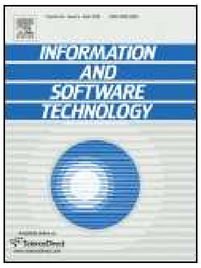
- **ACoTA'10**. J.A. Galindo, D. Benavides and *S. Segura*. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. *Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications (ACoTA'10).* Antwerp, Belgium, 2010.

IS'10. D. Benavides, *S. Segura*, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35 (6):615 - 636, 2010.

JCR IF: 1.96
CS-IS: 31/116 (26.7%)

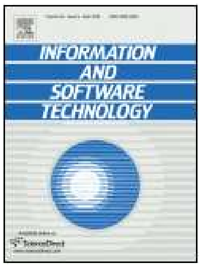
IET'10. *S. Segura*, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools: A test suite. *IET Software*, 2010 (*to appear*).

JCR IF: 0.65
CS-SE: 72/93 (77.4%)



IST'10-1. *S. Segura*, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation Testing on an Object-Oriented Framework: An Experience Report. *Information and Software Technology Special Issue on Mutation Testing*, 2010 (*to appear*).

JCR IF: 1.82
CS-SE: 19/93 (20.4%)



IST'10-2. *S. Segura*, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated Metamorphic Testing on the Analyses of Feature Models. *Information and Software Technology*, 53:245-258, 2011. (published on-line in 2010).

JCR IF: 1.82
CS-SE: 19/93 (20.4%)

A summary of our publications is presented in Figure §1.5. It classifies our contributions according to the year of publication (vertically) and the topics introduced in the previous section (horizontally). For each publication, a coloured circle is presented together with the acronym of the paper. The colour of the circle show the type of participation of the PhD candidate in the publication (first, second, third or later co-author). Large circles represent journal articles. A summary of the publications presented each year is showed at the bottom of the figure. As illustrated, the graph shows a clear definition of what has been our research path until now. We started working on the identification of operations of analysis on feature models and studying the performance of different solvers (APLE'06, GTTSE'06, SPLC'06, JISBD'06). The result of this tasks led us to provide automated support for the analysis of feature models in the form of the FaMa framework (VaMoS'07, OSSPL'07, JISBD'07, SPLC'08). Once the first prototype of the framework was developed, we identified the need to provide techniques and tools to support functional and performance testing in the analysis of feature models (SPLiT'08, ASPL'08-1, VaMoS'09, IS'10, IET'10, IST'10, IST'10-2). At the same time, we devoted some efforts to collaborate with other authors working in similar topics (ASPL'08-2, ViSPLE'08, ACoTA'10) and to study other promising ideas like the synergies between web services and product lines (SOAPL'07) or the automation of operations of modification on feature models (PROLE'07, GTTSE08).

Table §1.1 presents another view of our publication classified according to the place of publication.

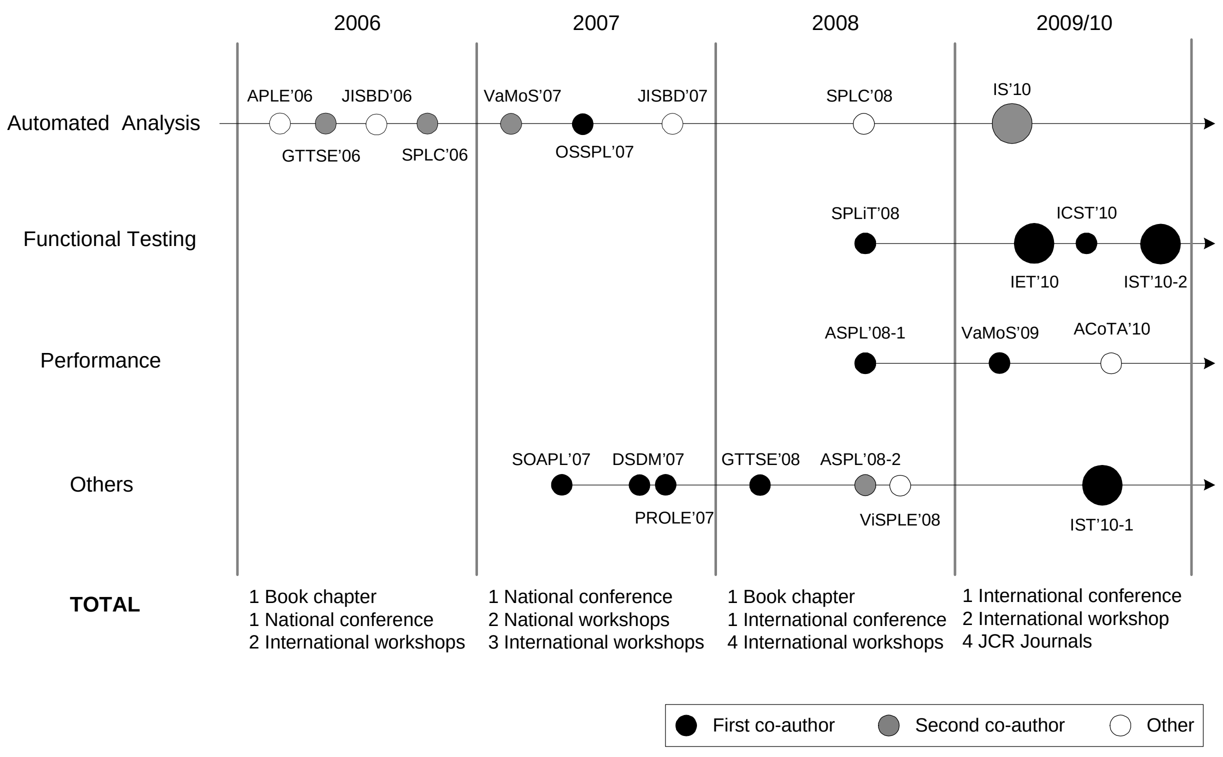


Figure 1.5: *Summary of publications grouped by topic and year.*

Category	Publications	Acronyms
JCR journals	4	IS'10, IET'10, IST'10-1, IST'10-2
Book chapters (LNCS)	2	GTTSE'06, GTTSE'08
International conferences	1	ICST'10
International workshops	11	APLE'06, SPLC'06, VaMoS'07, OSSPL'07, SOAPL'07, SPLiT'08, ASPL'08-1, ASPL'08-2, ViSPLE'08, VaMoS'09, ACoTA'10
National conferences	2	JISBD'06, PROLE,07
National workshops	1	DSDM'07
Tool demonstrations	2	JISBD'07, SPLC'08
Technical reports	7	-
Total	**30**	

Table 1.1: *Summary of publications grouped by place of publication.*

1.2.3 Citations

Table §1.2 summarizes the citations to our work in the context of software product line and feature modelling. Horizontally, we show the type of publication in which our work was cited. Vertically, we show the acronym of the paper cited as shown in the previous section. In total, our work has been cited 117 times (excluding self–citations) until October 2010 according to Google Scholar. The major number of citations (50) comes from international conferences which means that our work is fairly recognized by the international research community. The paper that has received more attention from the community with 29 citations is our contribution to the VaMoS workshop in 2007 (VaMoS'07) in which we presented the FaMa framework to the community. We may remark, however, that our literature review on the analysis of feature models (IS'10) has been cited 14 times since its publication in March 2010. This suggests that the number of citations of this work could grow significantly in the next years.

	GTTSE'06 [23]	JISBD'06 [20]	SPLC'06 [22]	APLE'06 [169]	VaMoS'07 [24]	SOAPL'07 [146]	GTTSE'08 [148]	SPLC'08 [168]	ASPL'08-1 [142]	VaMoS'09 [154]	IS'10 [21]	**Total**
JCR journals	0	2	1	1	1	0	0	0	0	0	0	**5**
Other journals	4	0	2	0	0	0	0	0	0	0	0	**6**
International conferences	1	7	6	1	13	1	5	4	3	1	8	**50**
International workshops	2	2	5	0	6	0	0	1	1	0	2	**19**
National conferences	0	1	0	0	0	2	0	1	0	0	0	**4**
National workshops	0	0	0	0	0	0	0	1	0	0	0	**1**
PhD dissertations	0	1	0	0	2	0	0	0	1	0	1	**5**
Others	5	4	5	1	7	1	1	0	0	0	3	**27**
Total	**12**	**17**	**19**	**3**	**29**	**4**	**6**	**7**	**5**	**1**	**14**	**117**

Table 1.2: *Summary of citations.*

1.2.4 Developed tools

The results of this thesis have been integrated into two tools developed in the context of our research group, namely:

- **FaMa Framework**. FaMa [58, 168] is an open source Java framework for the automated analysis of variability models. Since its release in 2007, FaMa has been used by at least 12 companies and universities in 7 different countries and it has been integrated into several third-party tools. The results of this thesis have endowed FaMa with new analysis components, performance optimization techniques and a new input format.
- **BeTTy Framework**. BeTTy [25] is an open source Java framework for functional and performance testing of feature model analysis tools. BeTTy is a direct result of our thesis and

integrates most of the contributions presented in this dissertation. A detailed description of the framework is presented in Appendix §A.

1.2.5 Research visits

During the development of this thesis, we visited the School of Information Systems, Computing and Mathematics at Brunel University (London, UK). From June to September of 2009, we worked closely with the Professor Robert M. Hierons, one of the world leading researchers in the software testing community. Among other results, we developed the prototype of the test data generator that we later presented in the Third International Conference on Software Testing, Verification and Validation (ICST'10) [151] and in the Information and Software Technology journal [153]. We also worked in the initial version of the experience report that was later published in the Information and Software Technology Special Issue on Mutation Testing [152]. Additionally, we developed the first version of our evolutionary algorithm for the generation of computationally–hard feature models. The development, refinement and application of this algorithm is one of the main tasks of our ongoing collaboration with the Professor Hierons.

1.3 Structure of this dissertation

This document is structured as follows:

Part I: Preface. It comprises this introduction chapter.

Part II: Background Information. In this part, we 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 present a summary of the most relevant analysis operations found in the literature and the works proposing automated support for them. This chapter is based on an extensive literature review on the analysis of feature models presented by the authors in the Information Systems journal [21]. In Chapter §4, we present the main concepts related to software testing used in the context of our dissertation.

Part III: Our Contribution. This part is the core of our dissertation and is organized in four chapters. In Chapter §5, we present the problems addressed in our thesis together with an analysis of current solutions emphasizing the gap filled by our contributions. In Chapter §6, we present a test suite for functional testing on the analysis of feature models. This chapter is based on a journal article presented by the authors in the IET Software journal [150]. In Chapter §7, we present an automated test data generator for the analysis of feature models based on metamorphic testing. This chapter is based on a journal paper presented by the authors in the Information and Software Technology journal [153]. Finally, in Chapter §8, we present a novel evolutionary algorithm for the generation of hard feature model to be used in performance testing of feature model analysis tools.

Part IV: Final Remarks. It consists of Chapter §9 in which we report our main conclusions and our plans for future research.

Part V: Appendices. In Appendix §A, we present the BeTTy framework, a tool that integrates most of our contributions in the field of testing of feature model analysis tools. A complete list of the mutation operators used in our evaluations is presented in Appendix §B. In Appendix §C, we present an experience report with the lessons learned from using mutation testing to evaluate the effectiveness of our automated test data generator. This appendix is based on a journal paper accepted for publication in the Information and Software Technology Special Issue on Mutation Testing [152]. In Appendix §D, we present the statistical analysis data from the experimental results obtained during the evaluation of our evolutionary algorithm for the generation of computationally–hard feature models. Finally, we clarify the meaning of the acronyms used throughout this dissertation in Appendix §E.

Part II

Background Information

Chapter 2

Feature models

Variability is the law of life,
and as no two faces are the same,
so no two bodies are alike (...)

William Osler, 1849–1919
Canadian Physician

Feature models were introduced twenty years ago as a way to model variability in software product lines. Since then, a number of extensions to the original notation has been proposed. In this chapter, we survey the most common notations for feature modelling providing some examples. In Section §2.1, we introduce feature models. Section §2.2 presents the classical notation of feature models also referred to as basic feature models. Cardinality-based feature models are presented in Section §2.3. Section §2.4 introduces the works proposing extending feature models with attributes. Finally, we summarize the main points of the chapter in Section §2.5.

2.1 Introduction

In contrast to traditional software development, software product line engineering focuses on building set of related products rather than individual products. This introduces a new dimension in the development process referred to as *variability management*, i.e. managing what is common and what is different across the products of a product line. Variability must be correctly managed at all levels. For instance, requirement engineers must be aware of which requirements are mandatory and which ones are optional. Similarly, design engineers must know what features are incompatible and which ones depends on each other. Also, testers must know which are the products that can be derived from the product line in order to design their testing plans. In this context, feature models [85] are one of the most common artefacts for variability management in software product lines. A feature model provides a compact representation of all the products of a product line in terms of features where a feature is any domain abstraction relevant for the stakeholder [164]. More specifically, a feature model is a tree-like structure and consists of:

- Nodes representing products features.
- Relationships between a parent feature and its child features.
- Cross-tree constraints that are typically inclusion or exclusion statements of the form : "If feature A is included, then feature B must also be included (or excluded)".

Feature models are recognized in the literature to be one of the most important contributions to software product line engineering [12, 48]. One of the main benefits of feature models is their simplicity making it understood by all stakeholders [86]. In fact, feature models are widely used during the whole product line development process in order to produce other assets such as requirements documents [73, 101], architecture definition [127] or pieces of code [13, 48].

Feature models were first introduced as a part of the Feature-Oriented Domain Analysis method (FODA) by Kang back in 1990 [85]. Since then, feature modelling has been widely adopted by the software product line community. However, many extensions to the original proposal have been presented and a consensus about a feature model notation has not been reached yet. The goal of this chapter is to provide an overview of the feature modelling notations that we will refer to throughout the rest of this dissertation.

2.2 Basic feature models

We classify as basic feature models those in which simple relationships between features are used. Next, we detail them:

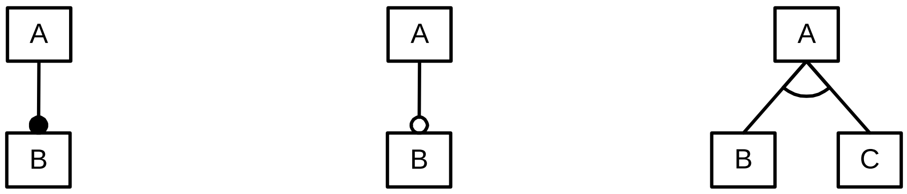
2.2.1 FODA feature models

Feature models were first introduced as a part of the Feature-Oriented Domain Analysis method (FODA) by Kang back in 1990 [85]. In this first approach, three kinds of relationships between features were proposed:

- **Mandatory.** A child feature has a mandatory relationship with its parent when the child is included in all products in which its parent feature appears. Mandatory relationships are generally modelled using a filled circle as shown in Figure §2.1(a). For instance, according to the feature model of Figure §2.4, it is mandatory that the software for mobile phones includes support for calls.

- **Optional.** A child feature has an optional relationship with its parent when the child can be optionally included in all products in which its parent feature appears. Optional relationships are generally represented using a empty circle as shown in Figure §2.1(b). For instance, support for multimedia devices (e.g. camera) is an optional feature in the feature model of Figure §2.4.

- **Alternative.** A set of child features have an alternative relationship with their parent when only one feature of the children can be selected when its parent feature is part of the product. Figure §2.1(c) depicts the usual visual representation for this relationship. As an example, according to the feature model of Figure §2.4, a mobile phone may use a Symbian or a WinCE operating system but not both in the same product.



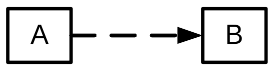
(a) *Mandatory.* (b) *Optional.* (c) *Alternative.*

Figure 2.1: *Feature relationships.*

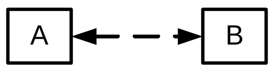
Notice that a child feature can only appear in a product if its parent feature does. The root feature is a part of all the products within the software product line. In addition to the parental relationships between features, two kinds of cross-tree constraints between features were identified in FODA. These are:

- **Requires.** If a feature A requires a feature B, the inclusion of A in a product implies the inclusion of B in such product. Requires constraints are commonly modelled using unidirectional arrows as shown in Figure §2.2(a). For instance, according to the feature model of Figure §2.5, mobile phones including games require Java support.

- **Excludes.** If a feature A excludes a feature B, both features cannot be part of the same product. This constraints are visually represented using bidirectional arrows as shown in Figure §2.2(b). As an example, the software product line represented by the model of Figure §2.5 rules out the possibility of offering support for MP3 and MP4 formats in the same product.



(a) *Requires.*



(b) *Excludes.*

Figure 2.2: *Cross-tree constraints.*

2.2.2 Feature-RSEB feature models

Griss *et al.* [71] proposed integrating the Feature-Oriented Domain Analysis method (FODA) into the so-called Reuse-Driven Software Engineering Business (RSEB). We call to this combination *Feature-RSEB*. RSEB is a use-case driven systematic reuse process in which variability is explicitly modelled by means of *variation points* and *variants*. In their approach, the authors proposed extending the FODA feature models with a new relationship between features. This is:

- **Or-relation.** A set of child features are said to have an *or-relation* with their parent when one or more of them can be included in the products in which its parent feature appears. Figure §2.3 depicts the common visual notation used for this type of relationship. For instance, according to the feature model of Figure §2.5, a mobile phone can provide connectivity support for Bluetooth, USB, wifi or any combination of the three.

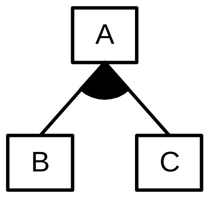


Figure 2.3: *Or relationship.*

2.2.3 Some examples

In order to clarify the concepts concerning basic feature models we present some examples. Figure §2.4 depicts a simplified feature model inspired by the mobile phone industry. The model illustrates how features are used to specify and build software for mobile phones. The software loaded in the phone is determined by the features that it supports. According to the model, all phones must include support for calls, messaging, and one operating system (i.e. OS), WinCE or Symbian (but not both of them). Furthermore, the software for mobile phones may optionally include support for multimedia devices such as camera, MP3 or both of them.

An extended version of the previous example is shown in Figure §2.5. It includes more features and cross-tree constraints (see Section §2.2.1). The requires cross-tree constraint implies

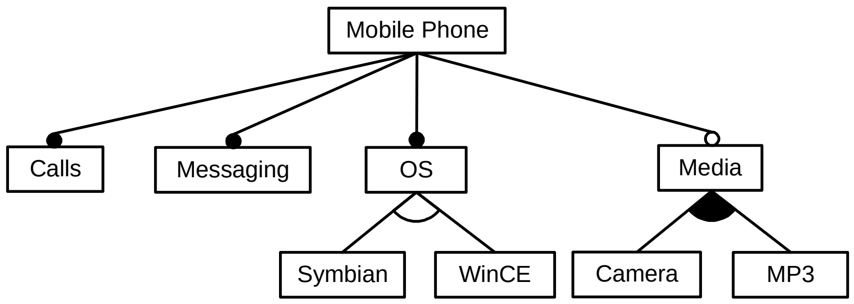


Figure 2.4: *Mobile phone feature model (without cross-tree constraints).*

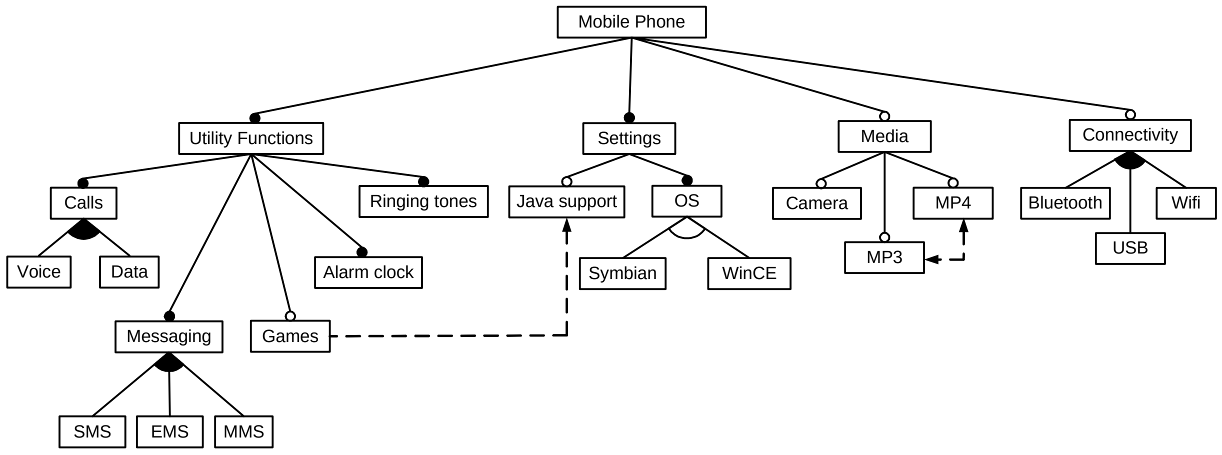


Figure 2.5: *Mobile phone feature model (with cross-tree constraints).*

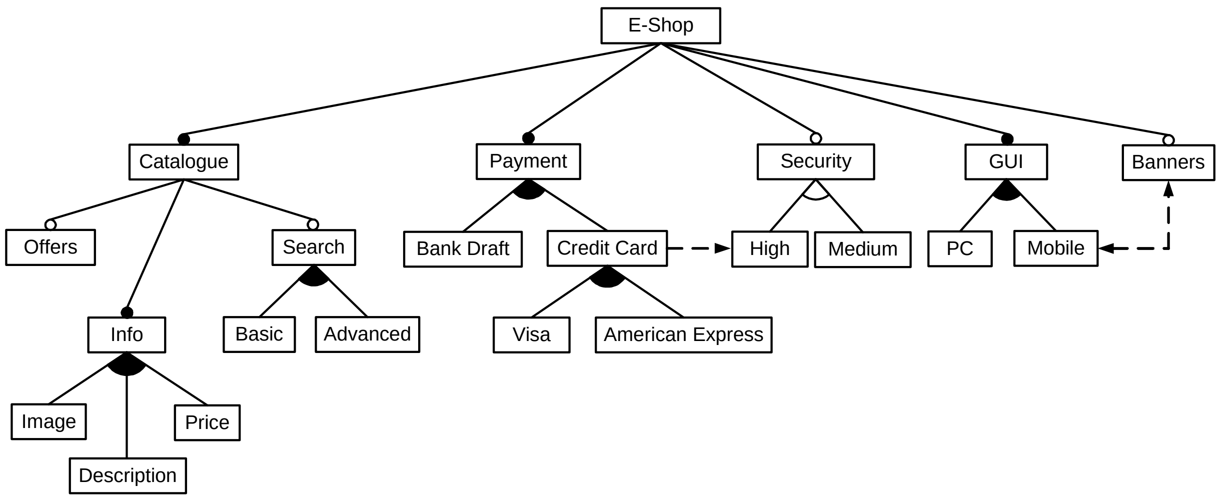


Figure 2.6: *E-shop feature model.*

that any phone including games must also include Java support. In a similar way, the excludes cross-tree constraint removes the possibility of providing support for MP3 and MP4 in the same product.

Figure §2.6 depicts a feature model representing the variability of a product line of on–line shopping systems. The products in the product lines must include mandatory features such as catalogue management or Graphic User Interface (GUI) and may include optional features such as banners or product search. Furthermore, shopping system must include an specific security policy (high or medium) and may include one or more payment modules. Finally, cross-tree constraints indicates that payment with creditcard requires a high security policy. Also, banners cannot be included in GUI for mobile devices.

2.3 Cardinality-based feature models

Some authors propose extending FODA feature models with UML-like multiplicities. An overview of those proposals and a clarifying example are next presented.

Riebisch et al.'s Proposal

According to Riebisch *et al.* the combination of mandatory and optional features with alternative and or-relationships could lead to ambiguities [134]. Motivated by this problem, the authors proposed the extension of FODA feature models with multiplicities (a.k.a. cardinalities) [134, 135]. In particular, they proposed using mandatory and optional relationships as in the original FODA proposal and replacing alternative and or-relationships with a new relationship:

- **Set relationship.** A set relationship relates a parent feature with a set of child features. Set relationships can include group cardinalities. A *group cardinality* is an interval,$\langle n..n' \rangle$, with n as lower bound and n' as upper bound limiting the number of child features that can be part of a product. For instance, in the set relationship of Figure §2.7, the number of child features that can be included in a product is limited to 1 or 2. This way, a set relationship with a group cardinality $\langle 1..1 \rangle$ is equivalent to the alternative relationship defined in the original FODA proposal (see Section §2.2.1). Similarly, a group cardinality of $\langle 1..N \rangle$, being N the number of children of the set relationship, is equivalent to an or-relation (see Section §2.2.2).

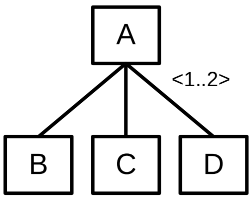


Figure 2.7: *Set relationship with group cardinality.*

Czarnecki et al.'s Proposal

Czarnecki *et al.* [49, 50] also proposed a number of extensions to the original feature modelling notation from FODA. One of those extensions is the usage of cardinalities. In their work, the authors proposed using the mandatory, optional and set relationships previously detailed. Additionally, they also introduced a new relationship:

- **Feature cardinality.** A feature cardinality is a sequence of intervals of the form $[n..n']$ whit n as lower bound and n' as upper bound. These intervals restrict the number of instances of the feature that can be part of a product. For instance, according to the feature cardinality showed in Figure §2.8, the number of instances of the feature B that can be included in a product is restricted to 1, 2 or 4. Notice that this relationship may be used as a generalization of the original mandatory and optional relationships defined in FODA. A feature cardinality of [0..1] would be equivalent to an optional relationship meanwhile a feature cardinality of [1..1] would be equivalent to a mandatory relationship.

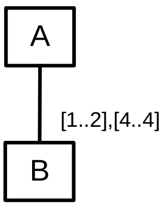


Figure 2.8: *Feature cardinality.*

As an example, Figure §2.9 depicts a cardinality–based version of the basic feature model presented in Figure §2.5. The model was built by replacing the alternative and or relationships of the original model with set relationships. Group cardinalities were adjusted conveniently to make the model represents the same products than the original model. In particular, alternative relationships were replaced by set relationships with group cardinality $\langle 1..1 \rangle$. Similarly, set relationships with group cardinality $\langle 1..N \rangle$ (being N the number of children) were used instead of or-relationships.

2.4 Extended feature models

Feature models are mainly used to manage functional variability in software product lines. However, several authors argue that in some contexts is also useful to add extra-functional information to the models. To this aim, several works propose adding attributes to the features [15, 16, 18, 49, 50, 57, 161]. These types of feature models in which extra information is included by means of attributes are usually referred in the literature as *extended, advanced or attributed feature models* [11, 12, 18, 51].

For instance, Figure §2.10 depicts a partial extended feature model using the notation proposed in [18]. As illustrated in the figure, an attribute mainly consists of:

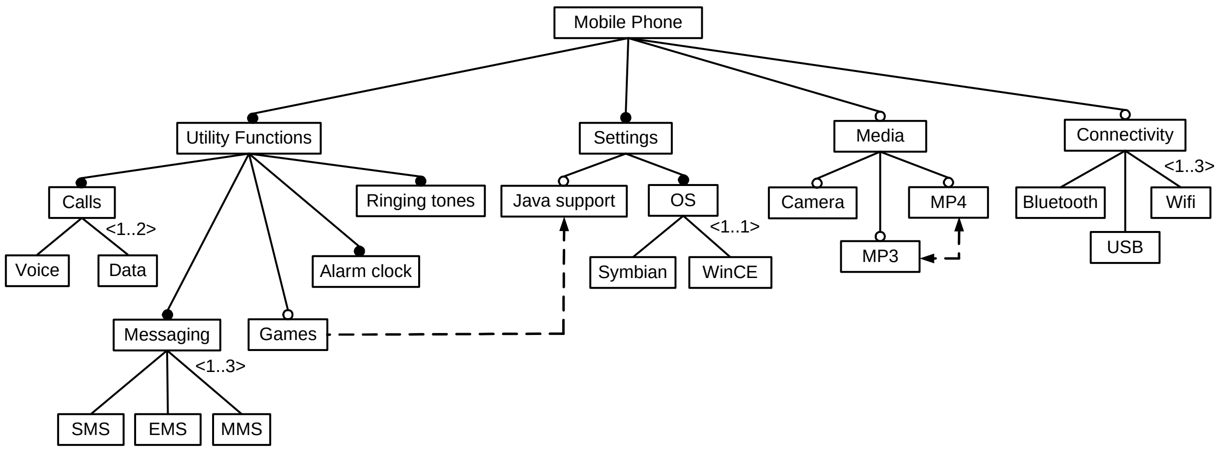


Figure 2.9: *Cardinality-based feature model.*

- *Attribute name.* Name or id of the attribute, e.g. memory.
- *Attribute domain.* It specifies the range of possible values for the attribute, e.g. real.
- *Attribute value.* The value of the attribute. This could be an specific value within the domain or an expression depending on the value of other attributes of the same or other features. A default value for the cases in which the feature is not selected could also be included.

Similarly to the relationships between features, relationships between attributes are also considered in extended feature models. This way, features could require or excludes other features depending on the value of any of their attributes. For instance, in the feature model of Figure §2.5, games could require an specific version of Java, e.g. *′*games *requires* Java.version *≥ 1.5′*.

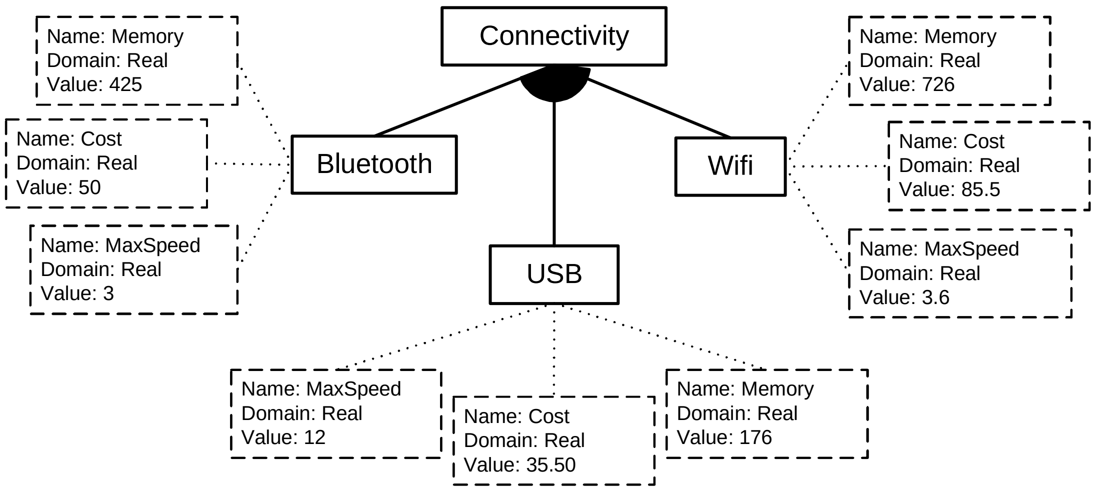


Figure 2.10: *Extended feature model.*

2.5 Summary

In this chapter, we have presented the most common feature modelling notations found in the literature. These notations are mainly extensions of the original feature model language presented in the FODA report 20 years ago. At the time of writing this dissertation, two main groups of feature model notations are used in the literature: basic feature models and cardinality-based feature models. The main difference is that cardinality-based feature models provide support for more complex relationships that the former one. As a possible complement to both of them, a third notation, extended feature models, propose adding extra functional information to the models by means of feature attributes. For a more extensive and rigorous survey of feature modelling languages we refer the reader to [140].

Chapter 3

Automated analysis of feature models

Computers are useless.
They can only give you answers.

Pablo Picasso, 1881–1973
Spanish Cubist painter

The automated analysis of feature models deals with the automated extraction of information from feature models. In this chapter, we overview the main contributions on the analysis of feature models in the last years. In Section §3.1, we introduce the main concepts related to the analysis of feature models. In Section §3.2, we present some of the most quoted analysis operations on feature models identified in the literature. A detailed description of the works proposing some kind of automated support for the analysis is presented in Section §3.3. Finally, in Section §3.4, we summarize the chapter and explain our main contributions in this field.

3.1 Introduction

Right from the introduction of feature models by Kang back in 1990, the manual manipulation of these was recognized as a difficult and error-prone task: *"it became clear that manual methods would not suffice, even in a relatively small example."* [85] (pag. 70). This mainly occurs due to the exponential number of possible feature combinations (i.e. products) in feature models. For instance, the E-shop feature model presented in Figure §2.6, with 22 features, represents more than 2,000 different products. To make matters worse, software product lines with thousands of features has been reported in the last years [12, 94, 160, 162]. In this situation, it is hard to imagine how to deal with feature models containing such a large number of features and complex dependencies between them without an appropriate automated tool support. Typical operations such as checking whether a feature model contains any errors become a tedious and time-consuming task under these circumstances. This makes the automated analysis of feature models to be considered as a relevant and challenging research topic in the software product line community [12].

The automated analysis of feature models can be defined as the computer–aided extraction of information from feature models. This extraction is mainly carried out in a two–step process depicted in Figure §3.1. Firstly, the input parameters, which necessarily include a feature model, are translated into a specific representation or paradigm such as propositional logic, constraint programming, description logic or ad–hoc data structures. Then, off–the–shelf solvers or specific algorithms are used to automatically analyse the intermediate representation of the input parameters and provide the result as an output.

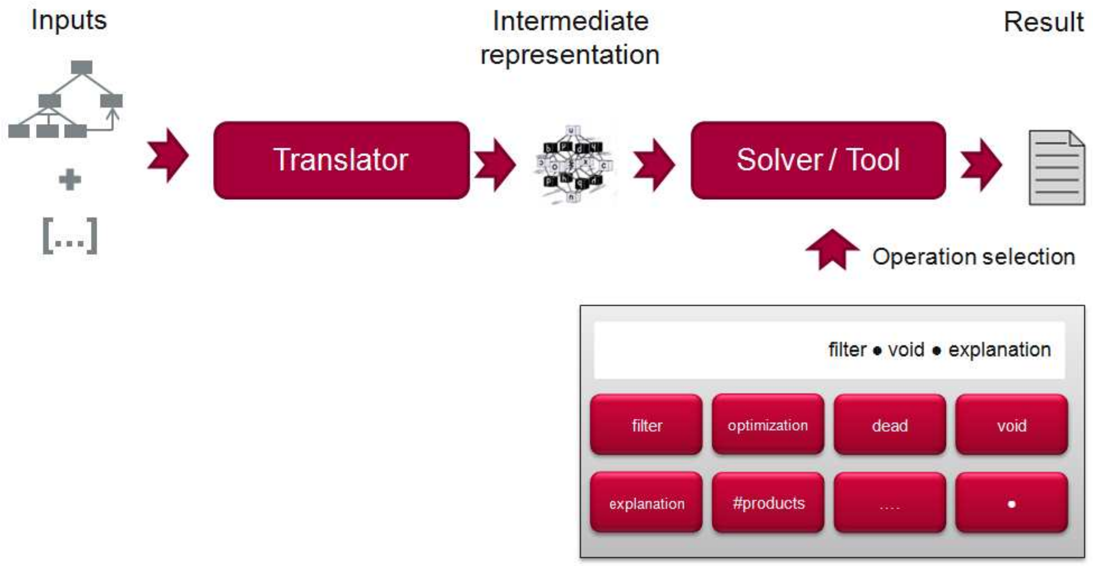


Figure 3.1: *Process for the automated analysis of feature models.*

The analysis of feature models is performed in terms of *analysis operations*. An operation takes a set of parameters as input and returns a result as output. From the analysis results obtained, marketing, technical and managerial decisions can be taken [15, 127, 166]. There exists a number of different operations that can be performed on feature models (e.g. finding out the number of products represented by a feature model). Similarly, there exist multiple proposals providing techniques, algorithms and tools to perform automatically some of those operations. The goal of this chapter is to review the state of the art in the context of the automated analysis of feature models. In particular, we first study the most relevant analysis operations found in the literature. Then, we analyse and compare the different proposals providing some kind of automated support for the analysis of feature models.

3.2 Analysis operations on feature models

During our research work, we performed an exhaustive revision of the state of the art on the analysis of feature models and identified 30 different analysis operations [21]. For the sake of simplicity and space constraints, we present in this section only those operations that we will refer to throughout the rest of the dissertation. We may remark that these operations are mainly those with more references in the literature according to our review.

In order to simplify the definition of the operations, we next clarify the meaning of various usually ambiguous terms found in the literature, namely:

- *Configuration*. Given a feature model with a set of features F, a configuration is a 2–tuple of the form (S,R) such that $S, R \subseteq F$ being S the set of features to be selected and R the set of features to be removed such that $S \cap R = \varnothing$. If $S \cup R = F$ the configuration is called *full configuration*. Alternatively, if $S \cup R \subset F$ the configuration is referred to as *partial configuration*. As an example, consider the model in Figure §3.2 and the full (FC) and partial (PC) configurations described below:

  $\mathsf{FC} = (\{\mathsf{MobilePhone}, \mathsf{Calls}, \mathsf{Screen}, \mathsf{Colour}\},$
  $\quad \{\mathsf{GPS}, \mathsf{Basic}, \mathsf{Highresolution}, \mathsf{Media}, \mathsf{Camera}, \mathsf{MP3}\})$
  $\mathsf{PC} = (\{\mathsf{MobilePhone}, \mathsf{Calls}, \mathsf{Screen}, \mathsf{Colour}\}, \emptyset)$

- *Product*. A product is equivalent to a full configuration where only selected features are specified and omitted features are implicitly removed. For instance, the following product is equivalent to the full configuration described above:

  $\mathsf{P} = \{\mathsf{MobilePhone}, \mathsf{Calls}, \mathsf{Screen}, \mathsf{Colour}\}$

For each operation, its definition, an example and possible practical applications are next presented.

3.2.1 Void feature model

This operation takes a feature model as input and returns a value informing whether such feature model is void or not. A feature model is *void* if it represents no products. The reasons that may make a feature model void are related with a wrong usage of cross–tree constraints, i.e. feature models without cross-tree constraints cannot be void.

As an example, Figure §3.3 depicts a void feature model. Constraint $\mathsf{C}-1$ makes the selection of the mandatory features B and C not possible, adding a contradiction to the model because both features are mandatory.

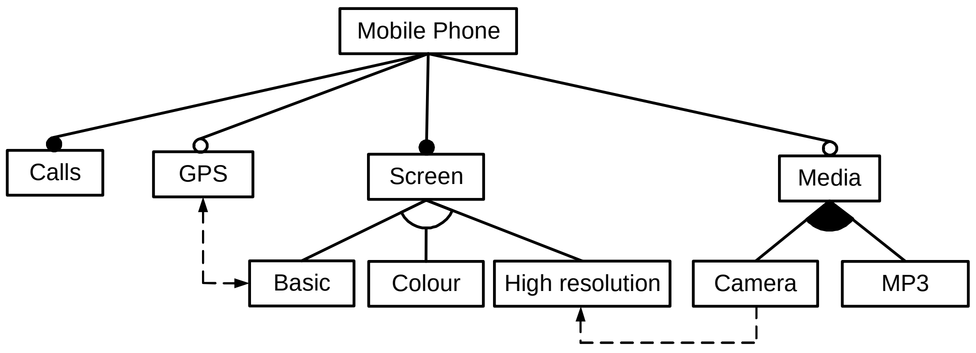


Figure 3.2: *Mobile phone feature model.*

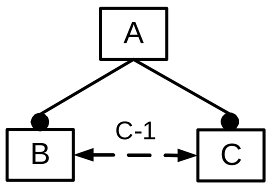


Figure 3.3: *A void feature model.*

The automation of this operation is especially helpful when debugging large scale feature models in which the manual detection of errors is recognized to be an error-prone and time–consuming task [11, 85, 165]. This operation is also referred to by some authors as *"model validation"*, *"model consistency checking"*, *"model satisfiability checking"*, *"model solvability checking"* and *"model constraints checking"*.

3.2.2 Valid product

This operation takes a feature model and a product (i.e. set of features) as input and returns a value that determines whether the product belongs to the set of products represented by the feature model or not. For instance, consider the products P_1 and P_2, described below, and the feature model of Figure §3.2.

$\mathsf{P}_1 = \{\mathsf{MobilePhone}, \mathsf{Screen}, \mathsf{Colour}, \mathsf{Media}, \mathsf{MP3}\}$
$\mathsf{P}_2 = \{\mathsf{MobilePhone}, \mathsf{Calls}, \mathsf{Screen}, \mathsf{Highresolution}, \mathsf{GPS}\}$

Product P_1 is not valid since it does not include the mandatory feature Calls. On the other hand, product P_2 does belong to the set of products represented by the model.

This operation may be helpful for software product line analysts and managers to determine whether a given product is available in a software product line. This operation is sometimes also referred to as *"valid configuration checking"*,*"valid single system"*, *"configuration consistency"*, *"feature compatibility"*, *"product checking"* and *"product specification completeness"*.

3.2.3 All products

This operation takes a feature model as input and returns all the products represented by the model. For instance, the set of all the products of the feature model presented in Figure §3.2 is detailed below:

$P_1 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Basic}\}$
$P_2 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Basic}, \text{Media}, \text{MP3}\}$
$P_3 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Colour}\}$
$P_4 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Colour}, \text{GPS}\}$
$P_5 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Colour}, \text{Media}, \text{MP3}\}$
$P_6 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Colour}, \text{Media}, \text{MP3}, \text{GPS}\}$
$P_7 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}\}$
$P_8 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}, \text{Media}, \text{MP3}\}$
$P_9 = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}, \text{Media}, \text{MP3}, \text{Camera}\}$
$P_{10} = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}, \text{Media}, \text{Camera}\}$
$P_{11} = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}, \text{GPS}\}$
$P_{12} = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}, \text{Media}, \text{MP3}, \text{GPS}\}$
$P_{13} = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}, \text{Media}, \text{Camera}, \text{GPS}\}$
$P_{14} = \{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Highresolution}, \text{Media}, \text{Camera}, \text{MP3}, \text{GPS}\}$

This operation may be helpful to identify new valid requirement combinations not considered in the initial scope of the product line. The set of products of a feature model is also referred to in the literature as *"all valid configurations"* and *"list of products"*.

3.2.4 Number of products

This operation returns the number of products represented by the feature model received as input. Note that a feature model is void iff the number of products represented by the model is zero. As an example, the number of products of the feature model presented in Figure §3.2 is *14*.

This operation provides information about the flexibility and complexity of the software product line [18, 51, 179]. A big number of potential products may reveal a more flexible as well as more complex product line. The number of products of a feature model is also referred to in the literature as *"variation degree"*.

3.2.5 Optimization

This operation takes a feature model and a so-called objective function as inputs and returns the product fulfilling the criteria established by the function. An objective function is a function associated with an optimization problem that determines how good a solution is.

This operation is chiefly useful when dealing with extended feature models where attributes are added to features. In this context, optimization operations may be used to select a set of features maximizing or minimizing the value of a given feature attribute. For instance, mobile phones minimizing connectivity cost in Figure §2.10 should include support for USB connectivity exclusively since it is the cheapest one.

3.2.6 Filter

This operation takes as input a feature model and a configuration (potentially partial) and returns the set of products including the input configuration that can be derived from the model. Note that this operation does not modify the feature model but filters the features that are considered.

For instance, the set of products of the feature model in Figure §3.2 applying the partial configuration (S, R) = ({Calls, GPS},{Colour, Camera}), being S the set of features to be selected and R the set of features to be removed, is:

$P_1 = \{$MobilePhone, Calls, Screen, Highresolution, GPS$\}$
$P_2 = \{$MobilePhone, Calls, Screen, Highresolution, Media, MP3, GPS$\}$

Filtering may be helpful to assist users during the configuration process. Firstly, users can filter the set of products according to their key requirements. Then, the list of resultant products can be inspected to select the desired solution [51].

3.2.7 Anomalies detection

A number of analysis operations address the detection of anomalies in feature models i.e. undesirable properties such as redundant or contradictory information. These operations take a feature model as input and return information about the anomalies detected. We identified five main types of anomalies in feature models reported in the literature [21]. In this chapter, we may emphasize the following two as the most quoted in the literature:

Dead features. A feature is *dead* if it cannot appear in any of the products of the software product line. Dead features are caused by a wrong usage of cross–tree constraints. These are clearly undesired since they give the user a wrong idea of the domain. Figure §3.4 depicts some typical situations that generate dead features (dead features are depicted in grey).

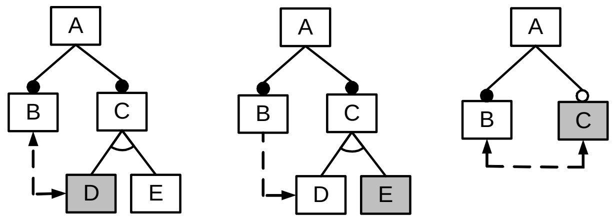


Figure 3.4: *Common cases of dead features.*

False optional features. A feature is *false optional* if it is included in all the products of the product line despite not being modelled as mandatory. Figure §3.5 depicts some examples of false optional features (false optional features are depicted in grey).

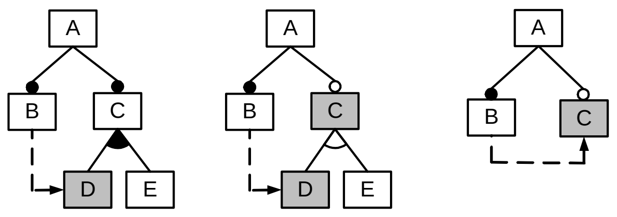


Figure 3.5: *Some examples of false optional features.*

3.2.8 Explanations

This operation takes a feature model and an analysis operation as inputs and returns information (so-called *explanations*) about the reasons of *why* or *why not* the corresponding response of the operation [172]. Causes are mainly described in terms of features and/or relationships involved in the operation and explanations are often related to anomalies. For instance, Figure §3.6 presents a feature model with a dead feature. A possible explanation for the problem would be *"Feature* D *is dead because of the excludes constraint with feature* B*"*. We refer the reader to [172] for a detailed analysis of explanation operations.

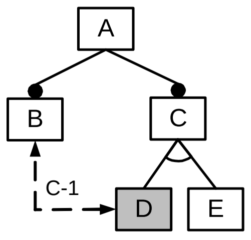


Figure 3.6: *Sample explanation.*

Explanations are a challenging operation in the context of feature model error analysis, (a.k.a. feature model debugging) [12, 165, 172]. In order to provide an efficient tool support, explanations must be as accurate as possible when detecting the source of an error, i.e. it should be minimal. This becomes an even more challenging task when considering extended feature models and relationships between feature attributes.

3.2.9 Commonality

This operation takes a feature model and a feature as inputs and returns the percentage of products represented by the model including the input feature. For instance, feature GPS in

Figure §3.2 is included in 6 out of the 14 products represented by the model. Therefore, its commonality can be calculated as $Comm(GPS) = 6/14 = 0.42$, which means that the feature appear in 42% of the products of the product line.

This operation may be used to prioritize the order in which the features are going to be developed [166] or to decide which features should be part of the core architecture of the software product line [127]. This operation can easily be generalized as described in [21].

3.2.10 Variability factor

This operation takes a feature model as input and returns the ratio between the number of products and 2^n where n is the number of features considered. In particular, 2^n is the potential number of products represented by a feature model assuming that any combination of features is allowed. The root and non-leaf features are often not considered. As an example, the variability of the feature model presented in Figure §3.2 taking into account only leaf features is:

$$\frac{N.Products}{2^n} = \frac{14}{2^7} = 0.0625$$

An extremely flexible feature model would be one where all its features are optional. For instance, the feature model of Figure §3.7 has the following variability factor:

$$\frac{N.Products}{2^n} = \frac{8}{2^3} = 1$$

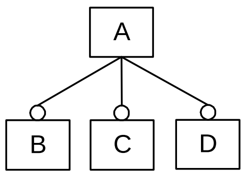


Figure 3.7: *Sample feature model with three optional features.*

The range of this indicator would depend on the features considered to calculate the factor. The feature model variability can be used to measure the flexibility of the feature model. For instance, a small factor means that the number of combinations of features is very limited compared to the total number of potential products.

3.3 Automated support

In this section, we present the proposals providing some kind of automated support for the analysis of feature models. To this purpose, we classify the works in four different groups according to the logic paradigm or method used to provide the automated support. In particular, we next present the group of approaches using *propositional logic*, *constraint programming*, *description logic*, and other contributions not classified in the former groups proposing *ad–hoc solutions*.

3.3.1 Propositional logic based analyses

A *propositional formula* consists of a set of primitive symbols or variables and a set of logical connectives constraining the values of the variables, e.g. $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

A *SAT solver* is a software package that takes as input a propositional formula and determines if the formula is satisfiable, i.e. there is a variable assignment that makes the formula evaluate to true. Input formulas are usually specified in *Conjunctive Normal Form* (CNF). CNF is a standard form to represent propositional formulas that is used by most of SAT solvers where only three connectives are allowed: $\neg, \wedge, \vee$. It has been proved that every propositional formula can be converted into an equivalent CNF formula [46]. SAT solving is a well known NP-complete problem [46], however, current SAT solvers can deal with big problems where in most of the cases the performance is not an issue [103].

Similarly, a *Binary Decision Diagram* (BDD) solver is a software package that takes a propositional formula as input (not necessarily in CNF) and translates it into a graph representation (the BDD itself) which allows determining if the formula is satisfiable and providing efficient algorithms for counting the number of possible solutions [32]. The size of the BDD is crucial because it can be exponential in the worst case. Although it is possible to find a good variable ordering that reduces the size of the BDD, the problem of finding the best variable ordering remains NP-complete [29].

The mapping of a feature model into a propositional formula can change depending on the solver that is used later for the analysis. In general, the following steps are performed: `i`) each feature of the feature model maps to a variable of the propositional formula, `ii`) each relationship of the model is mapped into one or more small formulas depending on the type of relationship, in this step some auxiliary variables can appear, `iii`) the resulting formula is the conjunction of all the resulting formulas of step `ii` plus and additional constraint assigning true to the variable that represents the root, i.e. `root` $\Leftrightarrow$ `true`.

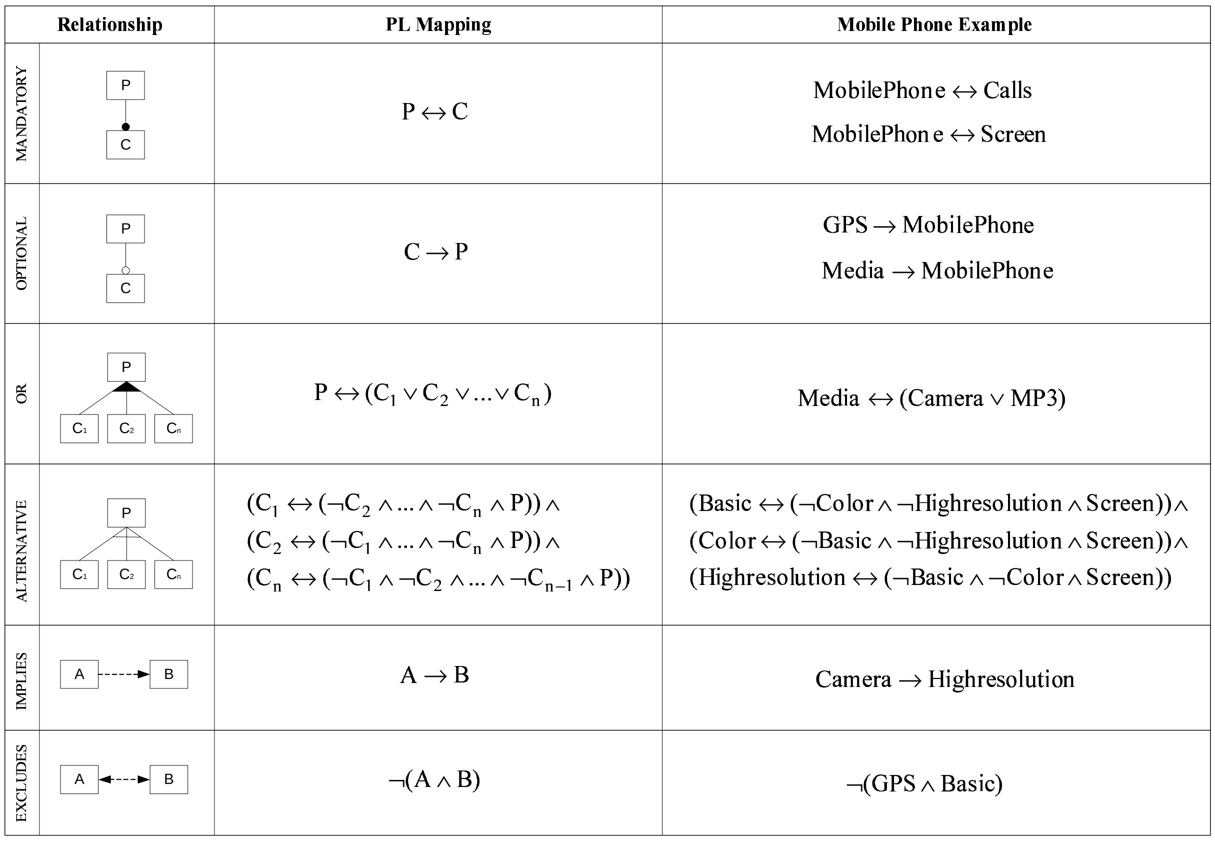
Concrete rules for translating a feature model into a propositional formula are listed in Figure §3.8. Also, the mapping of the mobile phone feature model presented in Figure §3.2 is presented. We may mention that the mapping of the propositional formulas listed in Figure §3.8 into CNF is straightforward (see [46]).

There are some works in the literature that propose the usage of propositional formulas for the automated analysis of feature models. In these studies the analysis is performed in two steps. Firstly, the feature model is translated into a propositional formula. Then, an off–the–shelf solver is used to automatically analyse the formula and subsequently the feature model. A summary of the solvers used for analysis is shown in Table §3.1.

To underline the most important contributions in terms of innovation with respect to prior work we may mention the following works: Mannion et al. [101, 102] were the first to connect propositional formulas and feature models. Zhang et al. [198] reported a method to calculate *atomic sets*, later explored by Segura [142]. Batory [11] showed the connections among grammars, feature models and propositional formulas, this was the first time that a SAT solver

Tool	Proposals
SAT Solver [138]	[11, 22, 24, 109, 142, 164]
Alloy [6]	[64, 163]
BDD Solver [80]	[22, 24, 51, 110, 142, 177, 178, 196, 199]
SMV [158]	[197, 198]
Not specified	[101, 102]

Table 3.1: *Propositional logic based tools used for the analysis of feature models.*



	Relationship	PL Mapping	Mobile Phone Example
MANDATORY	P, C	$P \leftrightarrow C$	$MobilePhone \leftrightarrow Calls$ $MobilePhone \leftrightarrow Screen$
OPTIONAL	P, C	$C \rightarrow P$	$GPS \rightarrow MobilePhone$ $Media \rightarrow MobilePhone$
OR	P, C1, C2, Cn	$P \leftrightarrow (C_1 \vee C_2 \vee ... \vee C_n)$	$Media \leftrightarrow (Camera \vee MP3)$
ALTERNATIVE	P, C1, C2, Cn	$(C_1 \leftrightarrow (\neg C_2 \wedge ... \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge ... \wedge \neg C_n \wedge P)) \wedge$ $(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{n-1} \wedge P))$	$(Basic \leftrightarrow (\neg Color \wedge \neg Highresolution \wedge Screen)) \wedge$ $(Color \leftrightarrow (\neg Basic \wedge \neg Highresolution \wedge Screen)) \wedge$ $(Highresolution \leftrightarrow (\neg Basic \wedge \neg Color \wedge Screen))$
IMPLIES	A, B	$A \rightarrow B$	$Camera \rightarrow Highresolution$
EXCLUDES	A, B	$\neg(A \wedge B)$	$\neg(GPS \wedge Basic)$

Figure 3.8: *Mapping from a feature model to propositional logic.*

was proposed to analyse feature models. In addition, a *Logic Truth Maintenance System* (a system that maintains the consequences of a propositional formula) was designed to analyse feature models. Sun et al. [163] proposed using Z, a formal specification language, to provide semantics to feature models. Alloy was used to implement those semantics and analyse feature models. Benavides et al.[22, 24, 142] proposed using a multi–solver approach where different solvers are used (e.g. BDD or SAT solvers) depending on the kind of analysis operations to be performed. For instance, they suggested that BDD solvers seem to be more efficient in general than SAT solvers for counting the number of products of a feature model. Mendonca et al. [110] also used BDDs for analysis and compared different classical heuristics found in the literature for variable ordering of BDDs with new specific heuristics for the analysis of BDDs representing feature models. They experimentally showed that existing BDD heuristics fail to scale for large feature models while their novel heuristics can scale for models with up to 2,000 features. Thüm et al. [164] presented an automated method for classifying feature model edits, i.e. changes in an original feature model, according to a taxonomy. The method is based on propositional logic algorithms using a SAT solver and constraint propagation algorithms. Yan et al. [196] proposed an optimization method to reduce the size of the logic representation of the feature models by removing irrelevant constraints. Mendonca et al. [109] showed by means of an experiment that the analysis of feature models with similar properties to those found in the literature using SAT solvers is computationally affordable. In [119], Nakajima proposed a semiautomated method to detect inconsistent fragments (which he calls *bugs*) in feature models. The author presented a mapping from a feature model to a propositional formula and a diagram–slicing algorithm for locating the inconsistencies. Some implementation details were given. Later, in [118], the authors proposed a more refined method to detect inconsistencies based on a Boolean Constraint Propagation (BCP) algorithm for non–clausal formulas. This method does not require to translate the models to CNF. This allows a direct translation from the inconsistencies found in the formulas to the feature tree. No implementation details were presented.

3.3.2 Constraint programming based analyses

A *Constraint Satisfaction Problem* (CSP) [173] consists of a set of variables, a set of finite domains for those variables and a set of constraints restricting the values of the variables. *Constraint programming* can be defined as the set of techniques such as algorithms or heuristics that deal with CSPs. A CSP is solved by finding states (values for variables) in which all constraints are satisfied. In contrast to propositional formulas, CSP solvers can deal not only with binary values (true or false) but also with numerical values such as integers or intervals.

A CSP solver is a software package that takes a problem modelled as a CSP and determines whether there exists a solution for the problem. From a modelling point of view, CSP solvers provide a richer set of modelling elements in terms of variables (e.g. sets, finite integer domains, etc.) and constraints (not only propositional connectives) than propositional logic solvers.

The mapping of a feature model into CSP can vary depending on the concrete solver that is used later for the analysis. In general, the following steps are performed: $\mathtt{i}$) each feature of the feature model maps to a variable of the CSP with a domain of 0..1 or TRUE, FALSE, depending on the kind of variable supported by the solver, $\mathtt{ii}$) each relationship of the model is mapped into a constraint depending on the type of relationship, in this step some auxiliary variables can appear, $\mathtt{iii}$) the resulting CSP is the one defined by the variables of steps $\mathtt{i}$ and $\mathtt{ii}$ with the corresponding domains and a constraint that is the conjunction of all precedent constraints plus and additional constraint assigning true to the variable that represents the root, i.e. $\mathtt{root} \Leftrightarrow \mathtt{true}$ or $\mathtt{root} == 1$, depending on the variables' domains.

Concrete rules for translating a feature model into a CSP are listed in Figure §3.9. Also, the mapping of our running example of Figure §3.2 is presented.

There are some works in the literature that propose the usage of constraint programming for the automated analysis of feature models. Analyses are performed in two steps. Firstly, the feature model is translated into a CSP. Then, an off–the–shelf solver is used to automatically analyse the CSP and subsequently the feature model. A summary of the solvers used for analysis is shown in Table §3.2.

Tool	Proposals
JaCoP [79]	[22–24, 142]
Choco [44]	[23, 192, 194]
OPL studio [124]	[17–19]
GNU Prolog [66]	[56]
SkyBlue [137]	[183, 184]
Not specified	[165, 167]

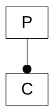
Table 3.2: *CSP based tools used for the analysis of feature models.*

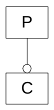
Benavides et al. were the first authors proposing the usage of constraint programming for analyses on feature models [17–19]. In those works, a set of mapping rules to translate feature models into a CSP were provided. Benavides et al. proposals provided support for the analysis of extended feature models (i.e. including feature attributes) and the operation of optimization. The authors also provided tool support [24, 168] and they compared the performance of different solvers when analysing feature models [22, 23, 142]. Trinidad et al. [165, 167] focused on the detection and explanation of errors in feature models based on Reiter's theory of diagnosis [133] and constraint programming. Djebbi et al. [56] proposed a method to extract information from feature models in terms of queries. A set of rules to translate feature models to boolean constraints were given. They also described a tool under development enabling the analysis of feature models using constraint programming. White et al. [194] proposed a method to detect conflicts in a given configuration and propose changes in the configuration in terms of features to be selected or deselected to remedy the problem. Their technique is based on translating a feature model into a CSP and adding some extra variables in order to detect and correct the possible errors after applying optimization operations. In [192], White et al. provided support for the analysis of multi–step configuration problems. Karatas et al. [88] presented a mapping from extended feature models to constraint satisfaction problem over finite domains. Their mapping supports both basic and cardinality–based feature models. In their work, the authors suggest that using their mapping the implementation of a number of analysis operations using an standard CSP solver would be straightforward but no implementation details were reported. Wang et al. [183, 184] presented a dynamic priority–based approach to fix inconsistencies in feature model efficiently. To that purpose, the authors proposed using the constraint hierarchy theory, a known practical theory used in the construction of graphical user interfaces. Implementation and evaluation details using the SkyBlue solver were presented.

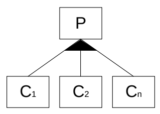
3.3.3 Description logic based analyses

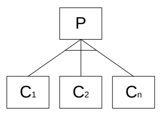
Description logics are a family of knowledge representation languages enabling the reasoning within knowledge domains by using specific logic reasoners [8]. A problem described in terms of description logic is usually composed by a set of concepts (a.k.a. classes), a set of roles (e.g. properties or relationships) and set of individuals (a.k.a. instances).

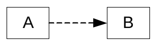
Relationship		CSP Mapping	Mobile Phone Example

MANDATORY		P = C	Mobilephone = Calls
			Mobilephone = Screen

OPTIONAL		if (P = 0)
		C = 0
			if (Mobilephone = 0)
			GPS = 0
			if (Mobilephone = 0)
			Media = 0

OR		if (P > 0)
		Sum(C1,C2,...Cn) in {1..n}
		else
		C1= 0, C2=0,…., Cn=0
			if (Media > 0)
			Sum(Camera,MP3) in {1..2}
			else
			Camera = 0, MP3 = 0

ALTERNATIVE		if (P > 0)
		Sum(C1,C2,...Cn) in {1..1}
		else
		C1= 0, C2=0,…., Cn=0
			if (Screen > 0)
			Sum(Basic,Colour,High resolution) in {1..1}
			else
			Basic = 0,Colour = 0, High resolution = 0

REQUIRES		if (A > 0) B>0	if (Camera > 0) High resolution > 0

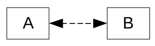
EXCLUDES		if (A > 0)
		B=0
			if (GPS > 0)
			Basic = 0

Figure 3.9: *Mapping from a feature model to CSP.*

A description logic reasoner is a software package that takes as input a problem described in description logic and provides facilities for consistency and correctness checking and other reasoning operations.

Several related works propose the usage of description logic to analyse feature models. Wang et al. [185] were the first to propose the automated analysis of feature models using description logic. In their work, the authors introduced a set of mapping rules to translate feature models into OWL-DL ontologies [52]. OWL-DL is an expressive yet decidable sub language of OWL [52]. Then, the authors suggested using description logic reasoning engines such as RACER [132] to perform automated analysis over the OWL representations of the models. In [186], the authors extended their previous proposal [185] with support for explanations by means of an OWL debugging tool. Fan et al. [59] also proposed translating feature models into description logic and using reasoners such as RACER to perform their analyses. In [1], Zaid et al. proposed using semantic web technologies to enable the analyses. They used OWL for modelling and the Pellet [128] reasoner for the analysis.

3.3.4 Other studies

There are some related works that are not classified in the former groups, namely: i) proposals in which the conceptual logic used is not clearly exposed and ii) works using ad–hoc algorithms, paradigms or tools for analysis.

Kang et al. mentioned explicitly the automated analysis of feature models in the original FODA report [85, pag. 70]. A prolog–based prototype was also reported. However, no detailed information was provided to replicate their prolog coding. After the FODA report, Deursen et al. [179] were the first authors proposing some kind of automated support for the automated analysis of feature models. In their work, they proposed a textual feature diagram algebra together with a prototype implementation using the ASF+SDF Meta-Environment [91]. Von der Massen et al. [180] presented Requiline, a requirement engineering tool for software product lines. The tool is mainly implemented by using a relational data base and ad–hoc algorithms. Later, Von der Massen et al. [181] proposed a method to calculate a rough approximation of the number of products of a feature model, which they call *variation degree*. The technique is described using mathematical expressions. In [9], Bachmeyer et al. presented *conceptual graph feature models*. Conceptual graphs are a formalism to express knowledge. They also presented an algorithm to enable the analysis of the models. Hemakumar [74] proposed a method to statically detect conditional dead features. The method is based on model checking techniques and incremental consistency algorithms. Mendonca et al. [106, 108] studied dependencies among feature models and cross–tree constraints using different techniques obtaining a noticeable improvement in efficiency. Gheyi et al. [65] presented a set of algebraic laws in feature models to check configurability of feature model refactorings. They used the PVS tool to do some analysis although this was not the main focus of the paper. White et al. [191] presented an extension of their previous work [193]. The same method was presented but giving enough details to make it reproducible since some details were missed in their previous work. The method is called *Filtered Cartesian Flattering* which maps the problem of optimally selecting a set of features according to several constraints to a *Multi–dimensional Multi–choice Knapsack Problem* and then they applied several existing algorithms to obtain efficient and near optimal solutions to the problem. Van den Broek et al. [176] proposed transforming feature models into generalised feature trees and computing some of their properties. A *generalised feature tree* is a feature model in which cross-tree constraints are removed and features can have multiple occurrences. Some algorithms and an executable specification in the functional programming language Miranda were provided. The strength of their proposal lied in the efficiency of the analysis operation. Fernandez et al. [60] proposed an algorithm to compute the total number of products on what they call *Neutral Feature Trees*, trees that allow complex cross-tree constraints. Computing the total number of products the authors were also able to calculate the *homogeneity* of a feature tree as well as the *commonality* of a given feature. They finally compared the computational complexity of their approach with respect to previous work.

3.4 Summary

In this chapter, we have introduced the main concepts related to the analysis of feature models. In particular, we have presented those analysis operations that are more relevant in the context of this dissertation. Also, we have introduced those related works proposing support for the analysis of feature models and we have classified them into four groups according to the main paradigm used for the analysis: propositional logic, constraint programing, description logic and ah–doc solutions.

During our research work, we have made several contributions to the field of automated analysis of feature models, namely: we proposed a technique for the detection of errors in feature models in an international workshop [169]. We have developed a tool supporting the analysis of feature models and we have presented it to the community in different workshop and conference papers [24, 168, 170]. We also presented a survey of the state of the art on the analysis of feature models in the XI Jornadas de Ingeniería del Software y Bases de Datos [20]. This survey was later extended as an extensive literature review published in the Information Systems journal [21].

Chapter 4

Software testing

Program testing can be used to show the presence of bugs,
but never to show their absence!

Edsger Wybe Dijkstra, 1930–2002
Dutch computer scientist, 1972 Turing Award

In this chapter, we present the concepts of software testing that are relevant to understand our contributions. In Section §4.1, we present an overview of the testing techniques used in our thesis. In Section §4.2, we present the basic testing procedure and introduce the oracle problem. Sections §4.3 and §4.4 present the testing techniques that we used to design and evaluate test data for functional and performance testing of feature model analysis tools. Finally, we summarize the chapter in Section §4.5.

4.1 Introduction

According to the IEEE Standard for software testing documentation, software testing is defined as "*The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.*" [78]. Software testing plays a key role in the development of software to evaluate whether the program meets its requirements, both functional and non-functional, and to gain confidence about the absence of faults in the program. In fact, it is acknowledged that most of the resources of a software project (more than 50%) are invested in the detection and correction of faults [14, 117]. Despite this massive investment, however, it is widely known that the complexity of software makes it impractical, often impossible, to detect all the faults of a program [14, 47, 117].

In this chapter, we introduce the testing techniques used in the context of our dissertation. Figure §4.1 classifies these techniques according to the purpose we used them for. As illustrated, we address both functional and performance testing on the analysis of feature models. Regarding functional testing, we used several classical techniques for the design of a manual test cases, namely: equivalence partitioning, boundary–value analysis, error guessing and pairwise testing. Also, we used a more sophisticated method, metamorphic testing, for the automated generation of test data. To measure the effectiveness of our contributions on functional testing, we checked their ability to detect artificial faults introduced in the programs using mutation testing. Regarding performance testing, we used an evolutionary algorithm for the generation of test data and a thorough comparison with random search for its evaluation.

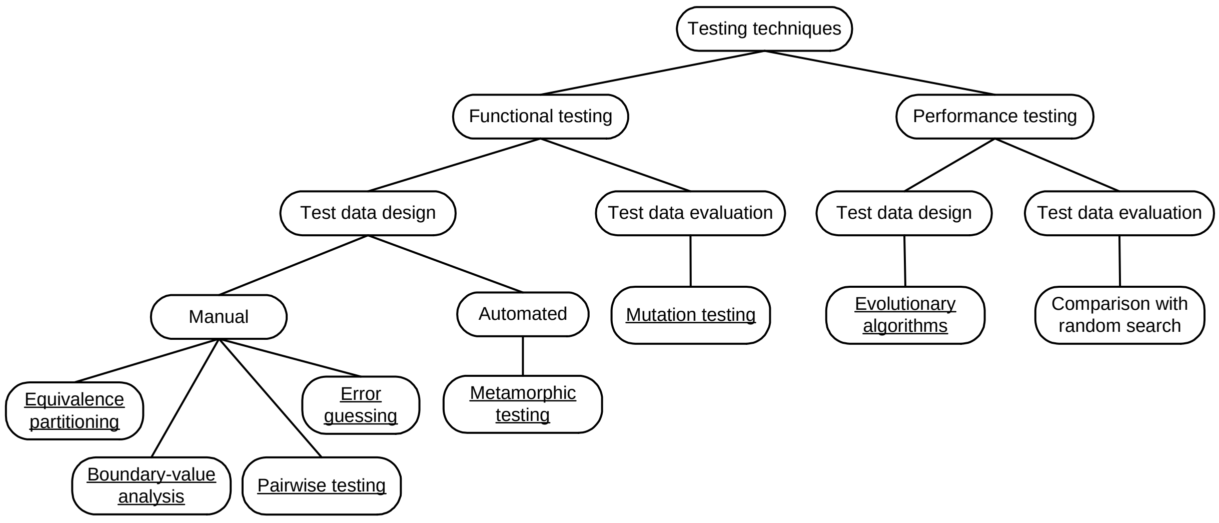


Figure 4.1: *Classification of testing techniques used in this dissertation.*

4.2 The testing procedure

Software testing is performed by means of test cases. A *test case* is a set of test inputs and expected outputs developed to verify compliance with a specific requirement [47, 117]. Figure §4.2 depicts the two basic steps of the testing procedure. First, the program is run with the selected inputs producing an output. Then, the output is examined to determine the program's correctness on the test data. The procedure by which testers can decide whether the output of a program is correct or not is referred to as *oracle* [189]. Typical oracles may include specification and documentation, statistical properties of the program or simply human being's judgment.

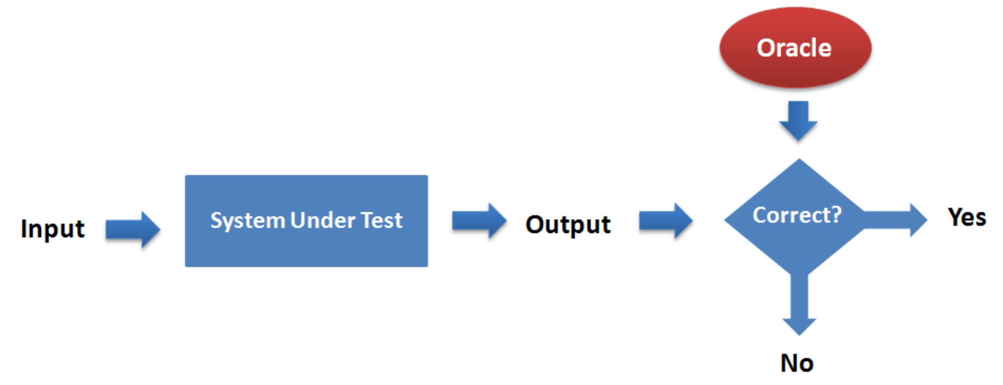


Figure 4.2: *Basic testing procedure.*

In some situations, the oracle is not available or it is too difficult to apply. This limitation is referred to in the testing literature as the *oracle problem* [200]. Consider, as an example, checking the results of complicated numerical computations (e.g. $\cos(x)$) or the processing of non-trivial outputs like the code generated by a compiler. Furthermore, even when the oracle is available, the manual prediction and comparison of the results are in many cases time–consuming and error–prone. In this context, Weyuker [189] defined a program to be *non-testable* if "*(1) there does not exist an oracle*" or "*(2) it is theoretically possible, but practically too difficult to determine the correct output.*"

Test cases may be grouped into so-called *test suites*. A variety of testing techniques has been reported to assist on the design of effective test cases, i.e. those that will find more faults with less effort and time [14, 47, 117]. These techniques can be classified according to multiple factors such as knowledge of the source code (black–box, white–box, gray–box) [28, 87, 117, 139, 201], source of information used (program–based, specification–based, interface–based) [28, 87, 201] or testing level (unit–level, integration–level, system–level). Also, there are different strategies to decide when the results of the test is adequate and testing should be stopped (so-called adequacy criteria). Common adequacy criteria are those that require the tests to cover a particular set of element in the program (e.g. branch coverage) or those that measures the adequacy of a test according to its ability to detect faults (e.g. mutation testing) [201].

4.3 Functional testing

Functional testing is intended to check whether a program satisfies its functional requirements, i.e. whether the program does what the user expect it to do. In the context of our dissertation, the goal of functional testing is to gain confidence in the correctness of the implementation of the analysis operation on feature models. In the following sections, we describe the techniques that we used to design and evaluate functional test data.

4.3.1 Test cases design

We will refer to the following test case design techniques for functional testing along this dissertation.

4.3.1.1 Equivalence partitioning

This technique is used to reduce the number of test cases to be developed while still maintaining a reasonable test coverage (i.e. the degree to which the test cases verifies the test requirements) [47, 117]. In this technique, the input domain of the program is divided into partitions (also called *equivalence classes*) in which the program is expected to process the set of data input in the same way. According to this testing approach, only one test case of each partition is needed to evaluate the behaviour of the program for the corresponding partition. The values within one partition are considered to be “equivalent” and thus selecting more than one value would not help to find new faults. Partitions can be created both in the input and the output domain.

As an example, consider a program that receives an integer as input representing a *month*. The valid range for the month is 1 to 12, representing January to December. This valid range represents a partition or equivalence class (see Figure §4.3). Similarly, there are two partitions of invalid ranges, one partition with those values less than 1 and another partition with those values greater than 12. The program is expected to process the values of each partition in an identical way. Thus, following the guidelines of equivalence partitioning, we could select just three input values, one from each partition (e.g. 0, 5, 14), to test the program effectively while reducing the number of test cases significantly.

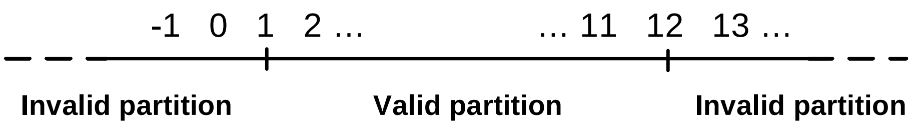


Figure 4.3: *Equivalence partitioning example.*

4.3.1.2 Boundary–value analysis

This technique is used to guide the tester when selecting inputs from equivalence classes [47, 117]. According to this technique, programmers usually make mistakes with the inputs values located on the boundaries of the equivalence classes. Thus, this method guides the tester to select those inputs located on and around the beginning and end of the equivalence partitions.

In the example presented in Figure §4.3, the boundaries values are 1 and 12. Using boundary–value analysis, test cases should be created with inputs that would fall on and to either side of each boundary. This would therefore result in three test cases per boundary, those with input values 0, 1, 2 and those with input values 11, 12, 13.

4.3.1.3 Error guessing

This is a software testing technique based on the ability of the tester to predict where faults are located according to its experience on the domain [47]. Using this technique, test cases are specifically designed to exercise typical error-prone points related to the type of system under test.

As an example, let us consider again the program that receive an integer representing a month as input. An experienced tester could have determined that the processing of holidays months (July and August) is error–prone. Thus, the tester could decide to design test cases with the inputs 7 and 8.

4.3.1.4 Combination strategies

The testing methods presented in previous sections support the selection of interesting values for each parameter of the system under test. Once that suitable values have been selected for each one of the input parameters of the system we must decide which combination of values should be used to design the test cases. *Combination strategies* determine the way in which input parameters are combined to form complete test cases. [70].

In order to illustrate the combination strategies, let us consider a program that receives three input parameters representing the Day of the Week (DW), Day of the Month (DM) and the Month (M). Table §4.1 shows the selected values for each parameter using equivalence partitioning as showed in Section §4.3.1.1. For simplicity, in the following we will refer to each value as P_i where P is the acronym of the parameter and i is the number of the column, e.g. $DM_2 = 24$. The way in which these values are combined is determined by the combination strategy used. These strategies can be mainly classified according to their coverage as follows:

Parameter	Values		
	1	2	3
DW	-2	4	8
DM	-3	24	32
M	0	6	13

Table 4.1: *Test values for a sample program processing a date.*

- **1-wise**. This is the simplest coverage criterion. It just requires that every value of every parameter is included in at least one test case in the test suite. For instance, the following combinations of inputs (determining test cases) fulfil this criterion: {(DW_1, DM_1, M_1), (DW_2, DM_2, M_2), (DW_3, DM_3, M_3)}.
- **2–wise**. Also known as 100% pair–wise. This criterion requires that every possible pair of any two values is included in at least one test case. For instance: {(DW_1, DM_1, M_1), (DW_1, DM_2, M_1), (DW_1, DM_3, M_1),(DW_2, DM_1, M_2),(DW_2, DM_2, M_2), (DW_2, DM_3, M_2), (DW_3, DM_1, M_3), (DW_3, DM_2, M_3), (DW_3, DM_3, M_3)}.

- **t-wise**. This is a generalization a 2–wise in which it is required that every combination of any t values is included in at least one test case. When $t = N$ being N the number of parameters of the program, the strategy is referred to as *N–wise*. In our example, the number of test cases that would be required to fulfil 3–wise coverage is $3^3 = 27$, i.e. all the combination of the three values for the three parameters.

The selection of a combination strategy requires considering the trade–off between coverage and simplicity. Hence, a higher coverage implies a higher number of test cases and more testing effort.

4.3.1.5 Metamorphic testing

Metamorphic testing [37, 189] was proposed as a way to address the oracle problem, i.e. the problem of determining whether a test output is correct or not. The idea behind this technique is to generate new test cases based on existing test data. The expected output of the new test cases can be checked by using so–called *metamorphic relations*, that is, known relations among two or more input data and their expected outputs. As a positive result of this technique, there is no need for an oracle and the testing process can be highly automated.

Consider, as an example, a program that compute the cosine function ($\cos(x)$). Suppose the program produces output -0.39 when run with input x=42 radians. An important property of the cosine function is $\cos(x) = \cos(-x)$. Using this property as a metamorphic relation, we could design a new test case with x=-42. Assume the output of the program for this input is 0.42. When comparing both outputs, we could easily conclude the program is not correct.

Metamorphic testing has shown to be effective in a number of testing domains including numerical programs [38], graph theory [39] or service–oriented applications [33].

4.3.2 Test cases evaluation

In order to measure the quality of our tests, we measured their ability to detect artificial faults introduced in the programs. This technique is called mutation testing and it is described next.

4.3.2.1 Mutation testing

Mutation testing [53] is a common fault–based testing technique that measures the effectiveness of test cases. More specifically, it measures the quality of test cases according to their ability to detect faults. The method starts by introducing simple faults into a program creating a collection of faulty versions, called *mutants*. The mutants are created from the original program by applying syntactic changes to its source code. Each syntactic change is determined by a so–called *mutation operator*. Mutation operators are specifically designed for the programming language or paradigm to be mutated. At the time of writing this dissertation, two main types of mutation operators are used in the literature: *traditional* and *class–level* operators [82]. The former are those mainly designed for procedural languages such as C or Fortran. These mainly mutate traditional programming features such as algebraic or logical operators (e.g. $i \Rightarrow$ i++).

```
int sum(int a, int b)
{
    int result = a + b;
    return result;
}
```

a) Original code

```
int sum(int a, int b)
{
    int result = a / b;
    return result;
}
```

b) Mutant

```
int sum(int a, int b)
{
    int result = a + b;
    return result++;
}
```

c) Equivalent mutant

Figure 4.4: *An example of program mutation.*

The latter are specifically designed to introduce faults affecting object–oriented features like inheritance or polymorphism (e.g. *super* keyword deletion). Figure §4.4(b) depicts an example of mutant created by changing the operator $+$ of the original program in Figure §4.4(a) into the operator $/$.

Once the mutants have been created, test cases are used to check whether the mutants and the original program produce different responses. If a test case distinguishes the original program from a mutant we say the mutant has been *killed* and the test case has proved to be effective at finding faults in the program. Otherwise, the mutant remains *alive*. Mutants that keep the program's semantics unchanged and thus cannot be detected are referred to as *equivalent*. The percentage of killed mutants with respect to the total number of them (discarding equivalent mutants) provides an adequacy measurement of the test suite called the *mutation score*.

Figure §4.4(c) depicts an example of equivalent mutant generated by adding a post–increment to the local variable result in the return instruction. Notice that this change will not affect the behaviour of the program since the variable will not be read anymore after that instruction. The automated detection of equivalent mutants is unfeasible because program equivalence is an undecidable problem [82]. This is currently the main barrier for the practical application of mutation testing. Several techniques have been proposed to alleviate this problem but their effectiveness is limited [75, 122, 141].

The underlying theory of mutation testing relies on two main hypothesis: the Competent Programmer Hypothesis and the Coupling Effect [53]. The former hypothesis assumes that programmers are competent and they introduce simple faults as those created by mutation. Hence, if a test case prove to be good at killing mutants it is also expected to be good at detecting real faults. The later hypothesis states that test cases that are able to detect simple faults like those of mutants will also implicitly distinguish more complex errors. This explains why only simple faults are introduced by mutation.

Mutation testing have been applied not only to common programming languages but also to other domains such as SQL, aspect–oriented programs, network protocols, web services, etc. Also, a number of tools for mutation testing are currently available on the Web. We refer the reader to [82] for a detailed survey on mutation testing.

4.4 Performance testing

Performance testing evaluates how well the program behaves under a particular workload. This type of testing may deal with a number of indicators such as response time of the application, memory consumption, availability, number of concurrent users, etc. In the context of this dissertation, we focus on those indicators that give a better idea of the efficiency of the analysis techniques under test, mainly execution time and memory consumption. In order to generate representative test data for performance testing, we used evolutionary testing. This technique is next described.

4.4.1 Evolutionary testing

The exhaustive search for test inputs is acknowledged to be unfeasible due to the size and complexity of the programs, there are simply too many inputs combinations [3]. In this context, evolutionary testing have proved to be a promising solution for the automated generation of test data for both functional [105] and non–functional properties [3]. *Evolutionary testing* refers to the usage of evolutionary algorithms to generate test data. Evolutionary algorithms use heuristics to find solutions to hard problems at an affordable computational cost [10]. For the generation of test data, these algorithms translate the test criteria into an objective function (also called fitness function) that is used to evaluate and compare the candidate solutions respect to the overall search goal. Using this information, the search is guided toward promising areas of the search space.

Each candidate solution in an evolutionary algorithm is referred to as *individual* or *chromosome* in analogy to the evolution of species in biological genetics where DNA of individuals is combined and modified along generations enhancing species through natural selection. Two of the main properties of evolutionary algorithms are that they are heuristic and stochastic. The former means that there is no guarantee of obtaining the global optimum for the optimization problem. The latter means that different executions of the algorithm with the same input parameters can produce different output, i.e. they are not deterministic. Despite this, evolutionary algorithms are among the most widely used optimization techniques being applied successfully in nearly all scientific and engineering areas by thousands of practitioners [10].

As an example, let us consider the design of a car as an optimization problem. A similar example was used to illustrate the working of evolutionary algorithms in [188]. Let us consider that the goal of the optimization problem is to find a car design that maximize speed and minimize fuel consumption. This problem is hard since a car is a highly complex system in which speed and fuel consumption depends on a number of parameters such as engine type, components as well as shape and body elements. Moreover, this problem is likely to have extra constraints like keeping the cost of the car under a certain value, making some designs unfeasible. In order to solve this problem using evolutionary algorithms, a fitness function providing numerical values associated to designs is needed. This fitness function drives the optimization process to our objective. Thus, designs that provide a better relation between fuel consumption and speed are expected to have better fitness values.

All the variants of evolutionary algorithms are based on a common working scheme shown in Figure §4.5. The basic steps of this scheme are:

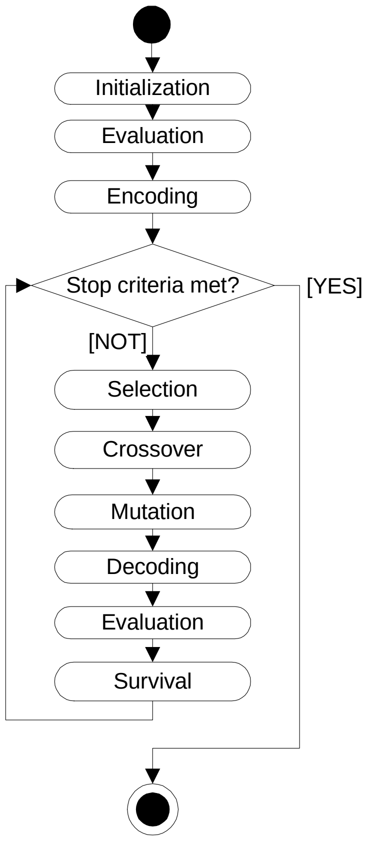


Figure 4.5: *General working scheme of an evolutionary algorithm.*

Initialization. The initial population (i.e. set of candidate solutions to the problem) is usually generated randomly. In our example, this could be done by choosing a set of random values for the design parameters of the car. Of course, the chances of finding optimal or near optimal car designs in this initial population are very small. However, promising values found at this step will be used to produce variants along the optimization process leading to better designs.

Evaluation. Next, individuals are evaluated using the fitness function that determines its optimality for the problem. The fitness function should be deterministic to avoid interferences in the algorithm, i.e. different calls to the function with the same set of inputs parameters should produce the same output. In our car example, a simulator could be used to provide the relation between fuel consumption and speed as fitness.

Encoding. In order to create offspring, individuals need to be *encoded* expressing its characteristics in a suitable form. In biological genetics, DNA encodes individual's characteristics on chromosomes that are used on reproduction and whose modifications produce mutants. For instance, classical encoding mechanisms on evolutionary algorithms are binary vectors encoding numerical values in genetic algorithms [10, Sec. C1.2] and tree structures encoding abstract syntax of programs in genetic programming [92]. In our car example, this step would imply to express design parameters of cars using some kind of data structure, e.g. binary vectors for each design parameter.

Stop criteria. Iterations on the remainder of the algorithm are performed until a termination criterion is met. Typical stop criteria are: reaching a minimum or average fitness value, maximum execution times of the fitness function, number of iterations of the loop (so-called generations) or number of iterations without improvements on the best individual found.

Selection. In the main loop of the algorithm (see Fig. §4.5), individuals are selected from current population in order to create new offspring. In this process, better individuals usually have more probability of being selected resembling the natural evolution where stronger individuals have more chances of reproduction. For instance, two classic selection mechanisms are roulette wheel and tournament selection [67]. When using the former, the probability of choosing an individual is proportional to its fitness determining the width of the slice of a hypothetic spinning roulette wheel. This mechanism is often modified assigning probability based on the position of the individuals in a fitness–ordered ranking (so-called rank-based roulette wheel). When using tournament selection, a group of n individuals is randomly chosen from the population and a winning individual is selected according to its fitness.

Crossover. These are the techniques used to combine individuals in some way and produce new individuals in an analogous way to biological reproduction. Crossover mechanisms depend on the encoding scheme used but standard mechanisms are available in literature for widely used encodings [10, Sec. C3.3]. For instance, two classical crossover mechanisms for vector encoding are one-point crossover [76] and uniform crossover [2]. When using the former, a random location in the vector is chosen as break point and portions of vectors after the break point are exchanged to produce offspring. When using uniform crossover, the value of each vector element is taken from one parent or other with a certain probability, usually 50%. Fig. §4.6(a) shows a high–level application of crossover in our example of car design. An F1 car and an small family car are combined by crossover producing a sports car. The new vehicle has some design parameters inherited directly of each parent such as number of seats or engine type and others mixed such as shape and intermediate size.

Mutation. At this step, random changes are applied to the individuals. Changes are performed with certain probability where small modifications are more likely than larger ones. This step is crucial to prevent the algorithm from getting stuck prematurely at a locally optimal solution. An example of mutation in our car optimization problem is presented in Fig. §4.6(b). The shape of a family car is changed by adding a back spoiler while the rest of its design parameters remain intact.

Decoding. In order to evaluate the fitness of new and modified individuals *decoding* is performed. For instance, in our car design example, data stored on data structures is transformed into a suitable car design that our fitness function can evaluate. It often happens that the changes performed in the crossover and mutation steps create individuals that are not valid designs or break a constraint, this is usually referred to as an *infeasible individual* [10], e.g. a car with three wheels. Once an infeasible individual is detected, this can be either replaced by an extra correct one or it can be repaired, i.e. slightly changed to make it feasible.

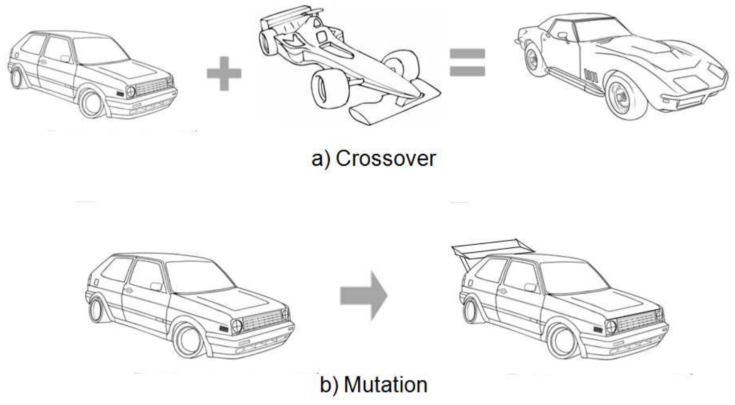


Figure 4.6: *Crossover and mutation in the search of an optimal car design.*

Survival. Finally, individuals are evaluated and the next population is conformed in which individuals with better fitness values are more likely to remain in the population. This process simulates the natural selection of the better adapted individuals that survive and generate offspring improving species.

4.5 Summary

In this chapter, we have presented the testing techniques that we will refer to during the rest of this dissertation. First, we have described those techniques that we used for the design and evaluation of test cases for the detection of faults in feature model analysis tools. Then, we have presented evolutionary testing, a technique based on natural evolution that we used for the generation of performance test data.

Part III
Our Contribution

Chapter 5
Motivation

Research is to see what everybody else has seen,
and to think what nobody else has thought.

Albert Szent-Gyorgyi, 1893–1986
Hungarian Biochemist, 1937 Nobel Prize for Medicine

In this chapter, we present the problems that we have addressed during our research work analyzing the current solutions and emphasizing the gap filled by our contributions. In Section §5.1, we motivate the chapter. The main problems addressed in this dissertation are presented in Section §5.2. In Section §5.3, we revise the current solutions found in the literature. In Section §5.4, we summarize and analyse current trends and explain the context of our contributions. Finally, we summarize the chapter in Section §5.5.

5.1 Introduction

The analysis of feature models is a thriving research topic that involves an increasing number of analysis operations, techniques and tools. The progress of this discipline is leading to an increasing interest in the quality of analysis solutions. However, the lack of specific testing mechanisms for functional and performance testing is becoming a major obstacle hindering the development of tools and affecting their reliability.

Supporting testing on the analysis of feature models require to overcome a number of problems described in the next section, among others: design of representative test data, development of test data generators overcoming the oracle problem, generation of hard problems, etc. Until now, most works on the analysis of feature models have either escaped the testing of their applications or used custom methods guided more by intuition than by well studied testing methods. This make testing results rarely rigorous and verifiable weakening the value and scope of contributions.

The final goal of this chapter is to motivate the need for specific methods for functional and performance testing on the analysis of feature models and introduce our contributions in this topic. These contributions are the results of the application of several classical and innovative testing techniques to the context of feature model analysis tools. We consider these contributions can promote the progress of the discipline both at a technical and a social level. From a technical standpoint, the usage of testing methods leads to a rigorous examination of the functional and performance results. From these results, the strengths and weaknesses of each proposal are identified helping researchers and practitioners to refine their solutions and to find new ways to improve them. From a social standpoint, testing results, especially those related to performance, can also promote the collaboration and communication among different researchers. As a result, these become more aware of the work carried out by their colleagues and collaborations among researchers with similar interests emerge naturally.

5.2 Problems

The main contributions of this thesis were motivated by the following problems, namely:

i. **FUNCTIONAL TESTING OF FEATURE MODEL ANALYSIS TOOLS**

Lack of representative test data. Feature model analysis tools deal with complex data structures and algorithms. This makes the implementation of analyses far from trivial and easily leads to faults increasing development time and reducing reliability of analysis solutions. Gaining confidence in the absence of faults in these tools is especially relevant since the information extracted from feature models is used all along the software product line development process to support important decisions [12]. In this context, the lack of specific test data is a major obstacle for engineers when trying to assess the functionality and quality of their analysis tools and remains as an open issue.

Automated generation of test data. The application of manual testing techniques rapidly becomes costly due to the complexity and high number of analysis operations on feature models. Automation is therefore desirable but difficult to apply due to the impossibility to determine the expected output of test cases, i.e. oracle problem. This occurs because the only way to check the correctness of the output of an analysis is by hand what in most of the cases is unfeasible. In fact, the analysis of feature models may fall into the category of software that Weyuker [189] classifies as *non–testable* because "*it is theoretically possible, but practically too difficult to determine the correct output*". The development of automated test data generators that overcome the oracle problem making the analysis of feature models testable is an open challenge.

ii. **PERFORMANCE TESTING OF FEATURE MODEL ANALYSIS TOOLS**

Automated generation of hard feature models. The number of works presenting performance results on the analysis of feature models has grown significantly. From the results obtained, the strengths and weaknesses of the applications are highlighted helping researchers to improve their solutions and identify new research directions. One of the main challenges in performance testing is to find hard problems that show the performance of tools in extreme situations, e.g. those maximizing execution time. Currently, hard feature models are generated randomly with a huge number of features and constraints. However, these only provide a rough idea of the behaviour of the tools with average problems and are not sufficient to reveal their real power. The generation of hard problems of realistic size to evaluate feature model analysis tools is an unexplored research topic.

5.3 Analysis of current solutions

In this section, we present the related works found in the literature. This is the result of an exhaustive literature review in which we found 25 related proposals published between November 2004 and October 2010. We may remark that we did not find any related work considering functional testing on the analysis of feature models. The works next presented are those reporting results about performance testing.

Benavides et al. [17–19] were the first presenting performance results of their CSP-based approach for the analysis of feature models. In their work, they used five sample feature models with a maximum of 15 features. Later, in [23], the authors presented a comparison of the performance of two CSP solvers for the analysis of feature models. They used two manually designed feature models and three models generated randomly with up to 52 features. This was the first work using random feature models and comparing the performance of different solvers. In [22], the authors extended their work by comparing the performance (execution time and memory consumption) of three analysis tools based on CSP, SAT and BDD. They used randomly generated feature model with up to 300 features and 25% of cross–tree constraints (with respect to the number of features).

Gheyi et al. [64] presented a propositional logic–based approach to analyse feature models. In their work, the authors reported execution times on several analysis on random feature models ranging in size up to 300 features.

Wang et al. [186] presented a case tool for checking product configurations and refactorings on feature models and evaluated it with a feature model of 1,000 features and 400 relationships. The authors mention that the model represents a large system but it is not clear whether it is real or it was generated randomly.

Hemakumar [74] proposed an algorithm based on boolean constraint propagation to detect conditionally dead features. As a part of his work, the author presented an implementation of his algorithm and reported the execution times for 11 feature models taken from the literature with up to 64 features.

Mendonca et al. [108] extended their previous work [106] formalizing their approach and providing some algorithms for dependency analysis in the context of collaborative product configuration. Execution times were reported for four random feature model with up to 2,000 features. Later, in [109], the authors presented experimental data showing that the some analysis of feature models are easily tractable by SAT–solvers. They analysed specific characteristics of this type of solvers and presented execution times using random feature models with up to 10,000 features. Feature models were generated with similar characteristics to those found in the literature.

Segura et al. [142] presented an algorithm and a prototype implementation to compute the atomic sets of a feature model. Executions times and memory consumption data for random feature models ranging in size up to 300 features and 25% of constraints were reported.

White et al. [190, 194] proposed a constraint-based diagnostic approach to detect inconsistencies in feature model configurations and repair them. Performance results, mainly execution times, were reported using random feature models ranging in size from 100 to 5,000 features and from 0% to 50% of cross-tree constraints. Later, in [191, 193], the authors proposed an algorithm to solve optimization problems on feature models. Experimental results with feature models with up to 10,000 were presented.

Zhang et al. [199] proposed a BDD–based approach to deal with the problem of analysis of feature models including feature cardinalities. They presented performance results comparing the proposal with their previous approach. In particular, execution times with random feature models between 10 and 1,000 features were reported.

Osman et al. [125, 126] proposed a knowledge base method to validate feature models and showed how it can be used to perform several analysis on the models. In their work, the authors presented some execution times with feature models with up to 20,000 features (referred to as variants in the paper) and 50% of cross-tree constraints.

Thüm et al [164] proposed automated support for classifying feature model edits, i.e. changes in an original feature model, according to a taxonomy. As a part of their work, the authors reported execution times using a realistic feature model taken from the literature with 287 features [93]. They also used randomly generated feature models (with up to 10,000 features and 10% of cross–tree constraints) with similar characteristic to those found in the literature.

Yan et al. [196] proposed a method and BDD–based tool to optimize the analysis of feature models by eliminating irrelevant features and constraints. Execution times for random feature models ranging in size from 100 to 1,900 features and 20% of cross-tree constraints as a maximum were reported.

She et al. [155] extracted a feature model from the Linux kernel in an attempt to find a hard and realistic problem to be used in the performance evaluation and benchmarking of feature model analysis tools. The model generated had 5,426 features. They also compared the generated model with those found in the literature. Galindo et al. [63] followed a similar approach. The authors propose modelling the Debian package repositories as variability models to obtain hard and realistic input models to evaluate the performance of analysis tools. The authors also suggest using the current techniques for the analysis of variability models to detect inconsistencies in the repositories.

Wan et al. [183, 184] proposed a dynamic-priority based approach to fixing inconsistent feature models. They also extend the constraint solver SkyBlue to implement a system that guide domain analysts in fixing inconsistencies. As a part of their work, they present the execution times of their tool when processing random feature model ranging in size up to 4,400 features and 10% of cross-tree constraints.

5.4 Discussion

From the analysis of related works, we conclude that functional testing is not addressed in the literature. This means that no evidences are shown about whether the implementation of the analysis operations are actually performing the right computation and about the absence of bugs in the applications. In contrast, most of the proposals focus on testing the performance of their tools reporting data about execution times and memory consumption. In other words, most authors try to answer the question *How efficient is my tool?* even before answering some more trivial ones: *Is my tool working right?, Does it contain any bug?*. We presume this lack of functional testing evidences occurs because it is simply much easier to perform basic performance evaluations than functional testing. In performance testing, the tool is run with an input feature model and performance data are recorded. In functional testing, however, two key challenges must be faced, namely: i) designing test data that provide a good test coverage, and ii) checking the correctness of outputs which is unfeasible in most of the cases for the analysis of feature models due to the oracle problem.

A key aspect in performance testing on the analysis of feature models is the type of subject problems used for the tests, i.e. test data. We found three main types of feature models used for experimentation: realistic, automatically generated feature models and those extracted from other domains with variability. By *realistic* models we intend those modelling real–world domains or a simplified version of them, e.g. e-shop feature model with 287 features [93]. Although there are reports from the industry of feature models with hundreds or even thousands of features [12, 94, 160], only a portion of them is typically published. This has led authors to generate feature models automatically to show the scalability of their approaches with large problems. These models are generated either randomly or trying to imitate the properties of the realistic models found in the literature. More recently, some authors have suggested looking for tough and realistic feature models into the open source community.

Fig. §5.1 summarizes the number of related works using realistic and automatically generated models as well as those extracted from other variability domains per each year. For each type of model, we also show the number of features of the largest feature model for each year. We may recall that the works included are those collected until October 2010. As illustrated, first works back in 2004 and 2005 used small realistic feature models in their experiments. However, since 2006, far more automatically generated feature models than realistic ones have been used. Regarding the size of the problems, there is a clear ascendant tendency ranging from the model

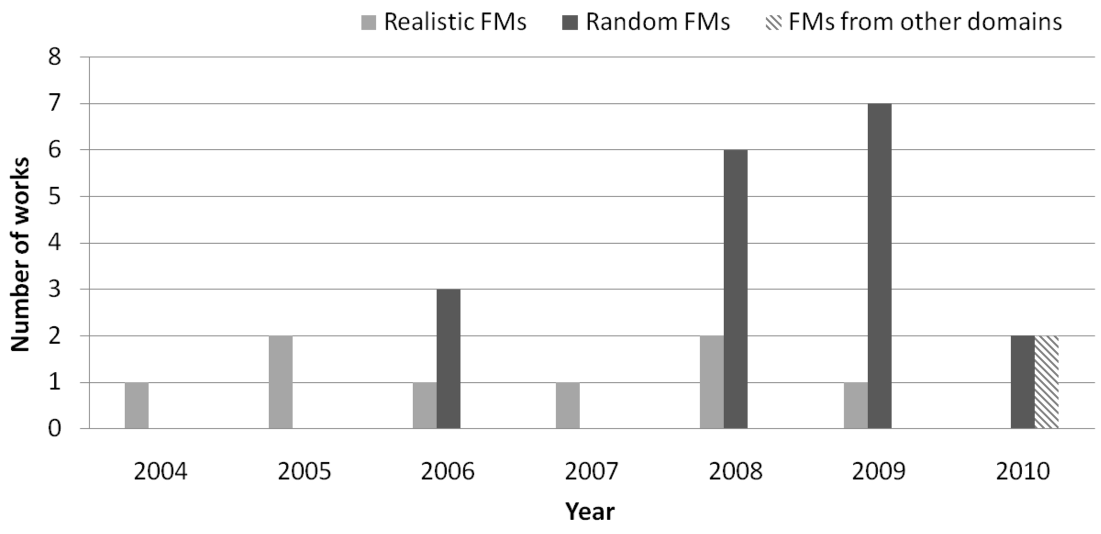


Figure 5.1: *Type and maximum size of the feature models used in performance testing.*

with 15 features used in 2004 to the model with 20,000 features used in 2009. These findings reflect an increasing concern to evaluate and compare the performance of different solutions using complex feature models. This also suggest that the only known mechanism to increase the complexity of the models is by increasing its size. A new trend, in 2010, propose to extract hard but realistic feature models from other domains with variability like those of open source operation systems.

Table §5.1 summarizes the related works found in the literature. Horizontally, the different proposals are presented grouped by the publication year. Vertically, we list the main contributions in the fields of functional and performance testing on the analysis of feature models. These contributions are mainly related to the type of test data (and associated tool support) proposed to perform the testing. For each row, the first cell marked with '+' indicates the year, depicted by the column, in which the contribution was first proposed. Later cells on each row are also marked with '+' to indicate that the contribution was already available. At first, the only test data used for performance testing were realistic models, usually invented. In 2006, the community started to use randomly generated feature models. This strategy is still the most popular among researchers as shown in Figure §5.1. In this thesis, we have provided the community with a new set of techniques and tools for both functional and performance testing on the analysis feature models. First, we have presented a carefully designed manual test suite, called FaMa Test Suite (FaMa TeS) for functional testing. Second, we have presented a framework, BeTTy, providing tool support for i) automated generation of test data for functional testing based on metamorphic testing, and ii) automated generation of hard feature models for performance testing using evolutionary algorithms.

	Benavides *et al.* [17]	Benavides *et al.* [18, 19]	Benavides *et al.* [22, 23], Gheyi *et al.* [64]	Wang *et al.* [186]	Hemakumar [74], Mendonca *et al.* [108, 110], Segura [142], White *et al.* [193, 194], Zhang *et al.* [199]	Mendonca *et al.* [109], Osman *et al.* [126], Thüm *et al.* [164], White *et al.* [190–192], Yan *et al.* [196]	Galindo *et al.* [63], She *et al.* [155], Wang *et al.* [183, 184]	**Our contributions**
	2004	2005	2006	2007	2008	2009	2010	2010
Functional testing								+
Manual test data								+
Test data generation								+
Performance testing	+	+	+	+	+	+	+	+
Realistic test data	+	+	+	+	+	+	+	+
Random test data			+	+	+	+	+	+
Hard test data generation								+

Table 5.1: *Summary of the related works.*

5.5 Summary

In this chapter, we have presented the main problems that motivated this dissertation. We have analysed the related literature on the analysis of feature models and observed that functional and performance testing is strongly necessary but poorly supported until now. We have also emphasized the value and originality of our main contributions, a manual test suite and a framework for functional and performance testing on the analysis of feature models.

Chapter 6

A test suite for the analyses of feature models

There are two ways to write error-free programs;
only the third one works.

Alan Jay Perlis, 1922–1990
American computer scientist, 1966 Turing Award

The analysis of feature models is usually automated using complex techniques of constraint programming, boolean formula satisfiability and description logic. Implementing analysis operations using these techniques is a time-consuming and complex task that easily leads to defects in analysis solutions. In this context, the lack of specific testing mechanisms is becoming a major obstacle hindering the development of tools and affecting their quality and reliability. In this chapter, we present FaMa Test Suite, a set of implementation–independent test cases to validate the functionality of feature model analysis tools. This is an efficient mechanism to assist in the development of tools, detecting faults and improving their quality. In order to show the effectiveness of our proposal, we evaluated the suite using mutation testing as well as real faults. Our results are promising and directly applicable in the testing of analysis solutions.

The remainder of the chapter is structured as follows: Section §6.1 introduces the problem tackled and the solution proposed. A detailed description of how we designed our test cases is presented in Section §6.2. Section §6.3 describes the adequacy evaluation and refinement of the suite. A summary of the refined suite and a brief discussion is presented in Section §6.4. Finally, we summarize the main points of our contribution in Section §6.5.

6.1 Introduction

The implementation of analysis operations on feature models is a hard task involving complex data structures and algorithms. This makes the development of analysis tools far from trivial and easily leads to errors increasing development time and reducing their reliability. Gaining confidence in the absence of defects in these tools is essential since the information extracted from feature models is used to support decisions all along the software product line development process [12]. However, the lack of specific testing mechanisms in this context appears as a major obstacle for engineers when trying to assess the functionality of their programs.

In this chapter, we present a set of implementation–independent test cases to validate the functionality of feature model analysis tools. Through the implementation of our test cases, faults can be rapidly detected assisting in the development of feature model analysis tools and improving their reliability and quality. For its design and evaluation, we used popular techniques from the software testing community to assist us on the creation of a representative set of input-output combinations. These test cases can be used either in isolation or as a suitable complement for further testing methods such as white–box testing techniques [47, 117] or automated test data generators [151]. As suggested by the testing literature, each test case was designed to reveal a single type of fault. This allows users to identify clearly the source of a fault once it has been detected. Briefly, we next describe the main characteristics of our suite:

- **Operations tested.** Current version of our suite, called FaMa Test Suite, addresses 7 out of 30 analysis operations on feature models identified in the literature [21]. These were selected for their extended use in the community of automated analysis and their heterogeneous nature.

- **Testing techniques.** Four testing techniques were used to design test cases, namely: equivalence partitioning, boundary-value analysis, pairwise testing and error guessing. A preliminary evaluation of the testing techniques to be used in our approach was presented in [143].

- **Test cases.** The suite is composed by 192 test cases. Each test case is designed in terms of the inputs (i.e. feature models and some other parameters) and expected outputs of the analysis operations under test.

- **Adequacy.** We evaluated the effectiveness of our suite using mutation testing and real faults as follows. Firstly, we generated hundreds of faulty versions (so-called mutants) of three open source analysis tools integrated into the FaMa framework. Then, we executed our suite against those faulty tools and check how many faults were detected by our test cases. As a result, the suite identified 96.1% of the faults showing the feasibility of our proposal. We then refined our suite until obtaining a score of 100%. Our refined suite also showed to be effective in detecting motivating faults found in the literature and in a recent release of the FaMa framework.

6.2 Test suite design

In this section, we describe how we designed the test cases that compose our suite. These mainly are input-output combinations specifically created to reveal failures in the implementations of analysis operations on feature models.

Creating test cases for every possible permutation of a program is impractical and very often impossible; there are simply too many input-output combinations [117]. Thus, as recalled by Pressman [130], the objective when testing is to *"design tests that have the highest likelihood of finding most errors with a minimum amount of time and effort"*. To assist us in the process, we evaluated a number of techniques reported in the literature [47, 117]. We focused on black-box techniques since we want our test cases to rely on the specification of the analysis operations rather than on specific implementations. In particular, we found four of these techniques to be effective and generic enough to be applicable to our domain, namely: equivalence partitioning, boundary–value analysis, error–guessing and pairwise testing. These techniques are described in Section §4.3.1.

For the design of the suite we followed four steps, namely: i) identification of the inputs and outputs of the analysis operations, ii) selection of representative instances of each type of input, iii) combination of previous instances in those operations receiving more than one input parameter, and iv) test cases report. Following, we detail how we carried out these steps.

6.2.1 Identification of inputs and outputs

The current version of our suite addresses 7 out of 30 analysis operations on feature models identified in the literature [21]. These operations are listed in Table §6.1 and fully described in Section §3.2. We selected these operations for its extended use in the community of automated analysis and their heterogeneous nature. In order to identify the type of the input/output parameters of the operations and avoid misunderstandings when interpreting their semantics, we used the formal definition of the operations proposed by Benavides [15]. Table §6.1 summarizes the operations in terms of their inputs and outputs. For the sake of simplicity, we assign an identificator to each operation to refer them along the chapter. As illustrated, inputs are composed of feature models, products and features. Outputs mainly comprise collections of products and features together with numeric and boolean values.

ID operation	**Operation**	**Inputs**	**Output**
VoidFM	Void FM	FM	Boolean value
ValidProduct	Valid product	FM, Product	Boolean value
Products	Products	FM	Collection of products
#Products	Number of products	FM	Numeric value
Variability	Variability	FM	Numeric value
Commonality	Commonality	FM, Feature	Numeric value
DeadFeatures	Dead features	FM	Collection of features

Table 6.1: *Analysis operations addressed in the suite.*

The feature modelling notation used in our suite corresponds to basic feature models, described in Section §2.2. We selected this notation for its simplicity and extended use in current feature model analysis tools and literature.

6.2.2 Inputs selection

In this section, we explain how we selected the inputs to be used in our test cases. For each type of input (i.e. feature models, products and features), we next describe the techniques used and how we applied them.

6.2.2.1 Feature models

We found two testing techniques to be helpful for the selection of a suitable set of input feature models, namely: equivalence partitioning and error guessing. We applied them as follows:

Equivalence partitioning. The potential number of input feature models is limitless. To select a representative set of these, we propose dividing input feature models into equivalence classes according to the different types of relationships and constraints among features, i.e. mandatory, optional, or, alternative, requires and excludes. This is a natural partition intuitively used in most proposals when defining the mapping from a feature model to a specific logic paradigm (e.g. constraint satisfaction problem) [11, 15, 18, 64, 102, 165, 186, 197]. Therefore, according to this technique, if a feature model with a single mandatory relationship is correctly managed by an operation, we could assume that those with more than one mandatory relationship would also be processed successfully.

To keep equivalence classes in a manageable level, we propose dividing the input domain into three groups of partitions as follows:

i. *Feature models including a single type of relationship or constraint*. Inputs from these partitions would help us to reveal failures when processing isolated relationships and constraints. For basic feature models, 6 partitions are created: feature models including mandatory relationships, optional, or, alternative, requires and excludes. Figure §6.1 (a) depicts the inputs we selected from each equivalence class using this criterion. Note that feature models with requires and excludes constraints also include an additional relationship (e.g. optional) to make them syntactically correct, i.e. sharing a common parent feature.

ii. *Feature models combining two different types of relationships or constraints*. We propose testing how analysis tools process feature model with multiple relationships by designing all the possible combinations of two of these, i.e. mandatory-optional, mandatory-or, etc. Following this criterion with basic feature models, we created a second group of 13 partitions. Figure §6.1 (b) presents the input models we took from each one of them. In general terms, feature relationships may appear in two main forms, child and sibling relationships (see Figure §6.2). According to our experience, inputs combining both types of relationships (highlighted in Figure §6.2) usually result in more complex constraints and therefore in more effective test cases. Thus, we selected inputs models from each

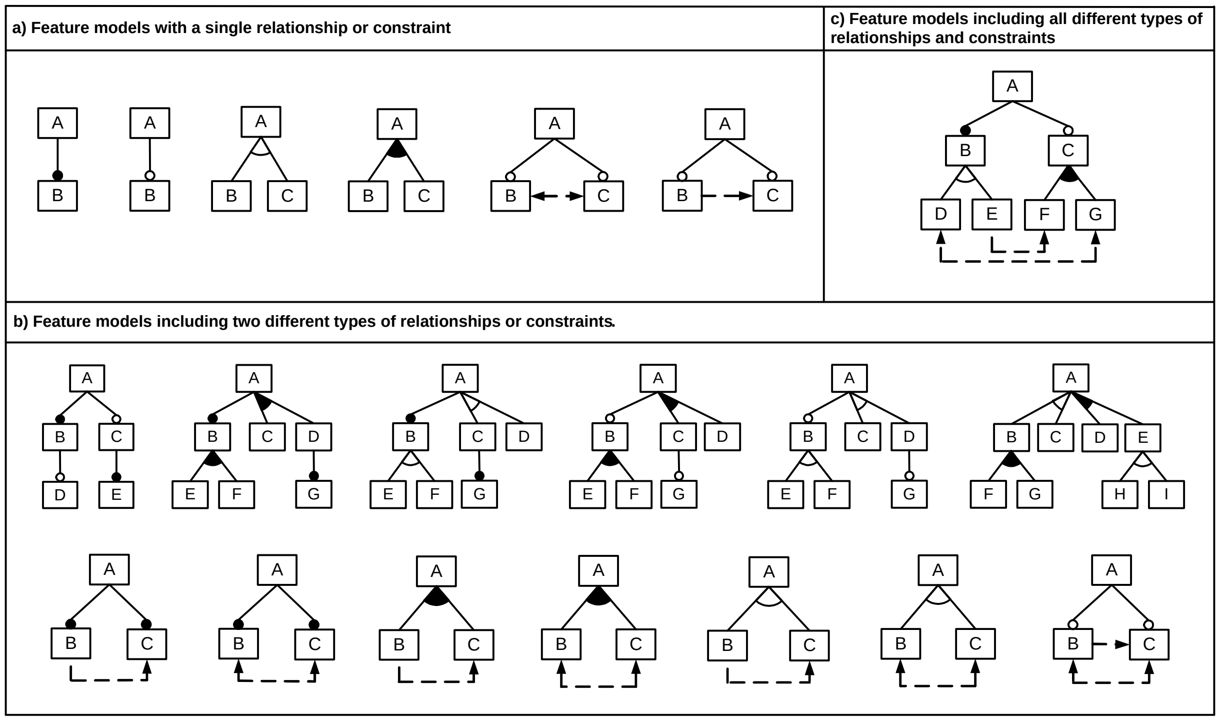

a) Feature models with a single relationship or constraint
c) Feature models including all different types of relationships and constraints
b) Feature models including two different types of relationships or constraints.
A
B
C
D
E
F
G
H
I

Figure 6.1: *Equivalence partitions on feature models.*

partition randomly but making sure these included both types of relationships, child and sibling relationships. For those input models including cross–tree constraints, we gave priority to simplicity and used sibling relationships exclusively.

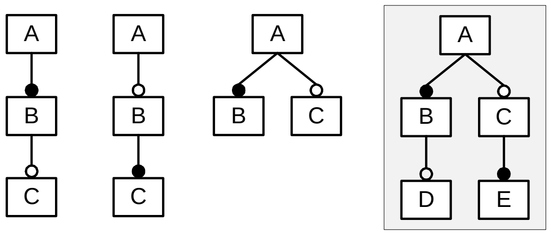


Figure 6.2: *Some possible combinations of mandatory and optional relationships.*

iii. *Feature models including three or more different types of relationships or constraints.* We finally propose creating a last partition including all those feature models not included in previous equivalence classes. Although this partition could be easily divided into smaller ones we did not find any evidence that justify such an increment in the number of test cases. Figure §6.1 (c) illustrates the random input feature model we took from this partition.

As a result of the application of this technique we got 20 feature models representing the whole input domain of basic feature models (see Figure §6.1). These were used as input in all the operations included in our suite with the exception of the operation *DeadFeatures* in which models derived from error–guessing were used.

Error guessing. Some common errors on feature models have been reported in the literature [15, 165]. As an example, Trinidad *et al.* [165] present some common problematic situations in which dead features appear. Figure §6.3 illustrate those situations (dead features are depicted in grey). Following the guidelines of error guessing, we propose using these models as suitable inputs to check whether dead features are correctly detected by the tools under test (i.e. operation for the detection of dead features). This way, we kept the number of input models for this operation in a reasonable level while still having a fair confidence in the ability of our tests to reveal failures.

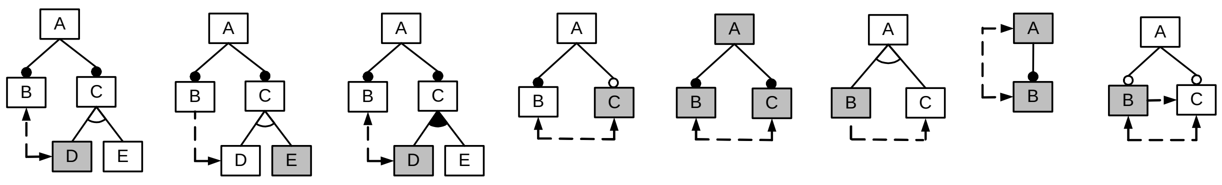


Figure 6.3: *Input feature models with dead features.*

6.2.2.2 Products

Products are used in the operation *ValidProduct* to determine whether a given product (i.e. set of features) is included in those represented by a feature model. For the selection of these inputs, we propose applying equivalence partitioning and boundary analysis as follows.

Firstly, we propose dividing input products into two equivalent partitions: *valid* and *non-valid* products. For each of these equivalence classes, inputs should be treated in the same way by the program under test and should produce the same answer. Then, we propose using boundary analysis for the systematic selection of input products from each partition. In particular, we suggest quantifying products according to the number of features they include. Then, we propose selecting those products on the “edges" of the partitions.

Figure §6.4 depicts an example of how we applied equivalence partitioning and boundary analysis for the selection of input products. As input feature models, we used those presented in previous section created using equivalence partitioning (Figure §6.1). For each input feature model, four inputs products were selected, two valid and two non-valid, as follows:

- VP_{min} *: valid product with the minimum number of features*. This product could be helpful to reveal failures caused by spare of erroneous constraints when processing minimal solutions of the problem. For instance, a failure using VP_{min}={A, B, D, E} may suggest that any of the other features (i.e. C, F or G) are erroneously treated as core features. A *core* feature is a feature that is part of all the products [172].
- VP_{max}: *valid product with the maximum number of features*. This product would allow testers to detect overconstrained representations of the model using large valid combination of features. As an example, a failure using VP_{max}={A, B, C, E, G} may suggest that

some spare constraint forcing the selection of D or F is not being fulfilled. Note that VP_{min} would still be helpful to detect, for example, whether non core features are erroneously included in all products.

- NVP_{min}: *non-valid product with one less feature than* VP_{min}. This product would be helpful to reveal failures caused by omitted or insufficiently constrained representations of the models. In the example, NVP_{min}={A, B, D} could help us to check whether the parent feature of an alternative relationship (B) can be erroneously included in a product without including any of its child features (E or F).
- NVP_{max}: *non-valid product with one more feature than* VP_{max}. This product would help us to check broken constraint caused by the selection of too many features. For instance, NVP_{max}={A, B, C, D, E, G} may reveal failures derived from including in a product more than one alternative subfeature (C and D). Once again, we would still need NVP_{min} to make sure that underconstrained solutions are detected.

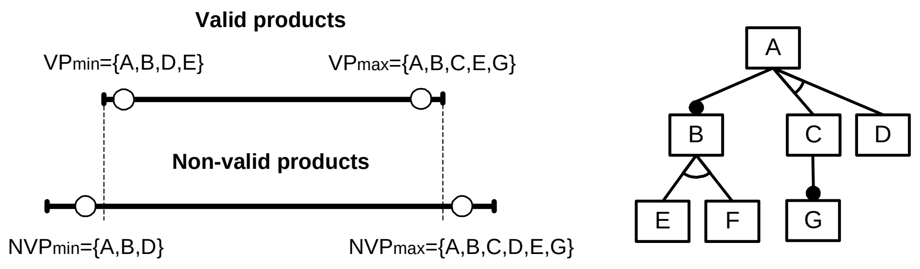


Figure 6.4: *Input products selection using partition equivalence and boundary analysis.*

We identified two special causes making a product to be non-valid, namely: i) not including the root feature (e.g. product {B, D, E} for the model in Figure §6.4), and ii) including non-existent features (e.g. product {A, B, D, E, **H**} for the model in Figure §6.4). These causes are applicable to all products independently of the characteristics of the input feature model. Therefore, we did not consider these situations for products NVP_{min} and NVP_{max}. Instead of this, we checked these problems in two separated test cases to be as accurate as possible when informing about failures.

6.2.2.3 Features

Given a feature model, the *Commonality* operation informs us about the percentage of products in which an input feature appears. For the selection of these input features, we propose applying equivalence partitioning and boundary analysis as follows.

To apply equivalence partitioning, we suggest focusing on the result space of the commonality operation. More specifically, we propose considering a single output partition: from 0% to 100% of commonality. Then, we propose applying boundary analysis and selecting those input features returning a value situated on the edges of the output partition. Figure §6.5 depicts an example of our proposal. For each input feature model, two input features are used, one with minimum commonality (G = 40%) and another one with maximum commonality (C = 80%). We intentionally exclude the root feature whose commonality is trivial (100%). We also included an additional test case to check the behaviour of the operation when receiving a non-existent feature as input (e.g. feature H for model of Figure §6.5).

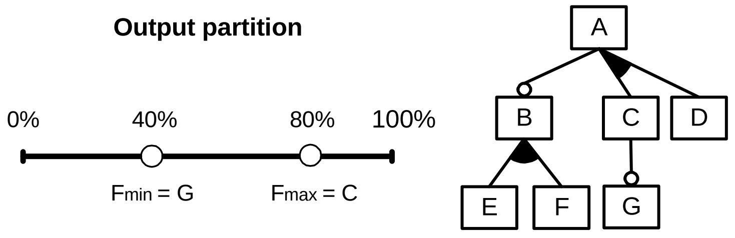


Figure 6.5: *Input features selection using partition equivalence and boundary analysis.*

6.2.3 Inputs combination

Combination strategies define ways to combine input values in those programs receiving more than one input parameter [70]. This is the case of two of the operations included in our suite: *ValidProduct* (it receives a feature model and a product) and *Commonality* (it receives a feature model and a feature). To create effective test cases for these operations, we used a pairwise combination strategy. That is, we created a test case for each possible combination of the two input parameters. Hence, our suite satisfies 2-wise coverage being 2 the maximum number of input received by the operations included on it.

As an example, consider the operation *ValidProduct* which receives two inputs: a feature model and a product. In previous section, we studied these inputs in isolation and selected representative values for them resulting in 20 feature models (see Figure §6.1) and 4 products (i.e. VP_{min}, VP_{max}, NVP_{min} and NVP_{max}). Using pairwise testing, we created a test case for each possible combination of them (i.e. 20*4 potential test cases). Note that some combination were not applicable (e.g. feature models without non–valid products) reducing the number of test cases for this operation to 60.

6.2.4 Test cases report

To conclude the design of our suite, we organized the selected inputs and their expected outputs into test cases; 180 in total. For their specification, we followed the guidelines of the IEEE Standard for Software Testing Documentation [78]. As an example, Table §6.2 depicts three of the test cases included in the suite. For each test case, an ID, description, inputs, expected outputs and intercase dependencies (if any) are presented. Intercase dependencies refer to the identifiers of test cases that must be executed prior to a given test case. As an example, test case *P-9* tries to reveals failures when obtaining the products of feature models including mandatory and alternative relationships. Note that test cases *P-1* (test of mandatory relationship in isolation) and *P-4* (test of alternative relationship in isolation) should be executed beforehand. Test case *C-28* exercises the interaction between requires and excludes constraints when calculating

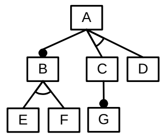
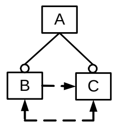
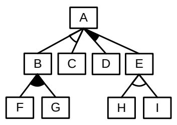
ID	Description	Input	Expected Output	Deps
P-9	Check whether the interaction between mandatory and alternative relationships is correctly processed.		{A,B,D,F}, {A,B,D,E}, {A,B,C,F,G}, {A,B,C,E,G}	P-1 P-4
C-28	Check whether the interaction between "requires" and "excludes" constraints is correctly processed. Input feature has minimum commonality.	 Feature=B	0%	C-2 C-5 C-6 C-7
VP-37	Check whether valid products (with a maximum set of features) are correctly identified in feature models containing or- and alternative relationships.	 P={A,B,D,E,F,G,H}	Valid	VP-5 VP-6 VP-7 VP-8 VP-9 VP-10

Table 6.2: *Three of the test cases included in the suite.*

commonality. Finally, test case *VP-37* is designed to reveal failures when checking whether an input product is included in those represented by a feature model including or- and alternative relationships. For a complete list of the test cases included in the suite we refer the reader to an external technical report [149].

6.3 Test suite evaluation and refinement

In this section, we first detail the results obtained when using mutation testing to evaluate and refine our suite. Then, we show the efficacy of our tests in detecting some real faults found in the literature and a recent release of the FaMa Framework.

6.3.1 Evaluation using mutation testing

In order to measure the effectiveness of our proposal, we evaluated the ability of our test suite to detect faults in the software under test (i.e. so-called fault-based adequacy criterion). To that purpose, we applied mutation testing on an open source framework for the analysis of feature models. We next present the experimental setup and the analysis of results.

6.3.1.1 Experimental setup

We selected FaMa Framework as a good candidate to be mutated. FaMa is an open source framework integrating different analysis components (so-called reasoners) for the automated analysis of feature models. It is integrated into the feature modelling tool MOSKitt [113] and it is currently being integrated into the commercial tool pure::variants [131] . Also, our familiarity with the tool made it feasible to use it for the mutations. In particular, we selected three of the analysis components integrated into the framework (so-called reasoners), namely: Sat4jReasoner v0.9.2 (using propositional logic by means of Sat4j solver [138]), JavaBDDReasoner v0.9.2 (using binary decision diagrams by means of JavaBDD solver [80]) and JaCoPReasoner v0.8.3 (using constraint programming by means of JaCoP solver [79]). Each one of these reasoners uses a different paradigm to perform the analyses and was coded by different developers, providing the required heterogeneity for the evaluation of our approach. For each reasoner, the seven analysis operations presented in Table §6.1 were tested.

For the mutations, we selected the key classes from each reasoner extending the framework and implementing the analysis capabilities. Table §6.3 shows size statistics of the subject programs including the number of classes selected from each reasoner, the total number of lines of code (LoC[†1]) and the average number of methods and attributes per class. The size of the 28 subject classes included in our study ranged between 35 and 220 LoC.

Reasoner	**LoC**	**Classes**	**Av Methods**	**Av Attributes**
Sat4jReasoner	743	9	6.5	1.8
JavaBDDReasoner	625	9	6.3	2.1
JaCoPReasoner	791	10	6.3	2.3
Total	**2,159**	**28**	**6.4**	**2.1**

Table 6.3: *Size statistics of the three subject reasoners.*

To automate the mutation process, we used MuClipse Eclipse plug-in v1.3 [157]. MuClipse is a Java visual tool for mutation testing based on MuJava [95]. It supports a wide variety of operators and can be used for both generating mutants and executing them in separated steps. Despite this, we still found several limitation in the tool. On the one hand, the current version of MuClipse does not support Java 1.5 code features. This forced us to make slight changes in the code, basically removing annotations and generic types when needed. On the other hand, we found the execution component provided by this and other related tools to not be sufficiently flexible, providing as a result mainly mutation score and lists of alive and killed mutants. To address our needs, we developed a custom execution module providing some extra functionality including: `i`) custom results such as time required to kill each mutant and number of mutants generated by each operator, `ii`) results in Comma Separated Values (CSV) format for its later processing in spreadsheets, and `iii`) filtering capability to specify which mutants should be considered or ignored during the execution. For the evaluation of the suite using MuClipse, we followed three steps, namely:

i. *Reasoners testing.* Prior to their analysis, we made sure the original reasoners passed all the test cases in our suite.

[†1]LoC is any line within the Java code that is not blank or a comment.

ii. *Mutants generation.* We applied all the traditional mutation operators available in MuClipse, a total of 15. These mainly mutate common programming languages features such as arithmetic (e.g. `++,-`) and relational operators (e.g. `==,!=`). Specific mutation operators for object–oriented code were discarded to keep the number of mutants manageable. Once generated, we manually discarded those mutants affection portions of the code not related to the analysis of feature models and therefore not addressed by our test suite (e.g. exception handling). A list of the mutation operators used in our evaluation is presented in Appendix §B.

iii. *Mutants execution.* For each mutant, we ran our test cases using JUnit [84] and tried to kill it. We may remark that the functionality of each operation was scattered in several classes. Some of these were reusable being used in more than one operation. Mutants on these reusable classes were evaluated separately with the test data of each operation using them for more accurate mutation scores, i.e. some mutants were executed more than once. As a result, the number of executed mutants was higher than the number of generated mutants. Equivalent mutants were manually identified and discarded after each execution.

6.3.1.2 Analysis of results

Table §6.4 shows information about the number of generated mutants. Out of the 760 generated mutants, 103 of them (i.e. 13.5%) were identified as semantically equivalent. In addition to these, we manually discarded 87 mutants (i.e. 11.4%) affecting other aspects of the program not related to the analysis of feature models and therefore not considered in our test suite. These were mainly related to the computation of statistics (e.g. execution time) and exception handling.

Reasoner	**Mutants**	**Equivalent**	**Discarded**
Sat4jReasoner	262	27	47
JavaBDDReasoner	302	28	37
JaCoPReasoner	196	48	3
Total	**760**	**103**	**87**

Table 6.4: *Mutants generation results.*

Table §6.5 depicts the results obtained when using our test suite to kill the mutants in the FaMa reasoners. For each operation and reasoner, the total number of mutants executed, number of alive mutants and mutation score are presented. The detection of dead features was tested only in JaCoPReasoner since this was the only reasoner implementing it. As illustrated, the operations *VoidFM* and *ValidProduct* produced the lowest scores and higher number of alive mutants. We found that mutants on these operations required input models to have a very specific pattern in order to be killed and therefore were harder to detect than mutants in the rest of operations. Average mutation scores in the three reasoners ranged between 94.4% and 100%. In total, our test suite was able to kill 1390 (96.1%) out of the 1445 mutants executed showing the effectiveness of the suite.

Operation	Sat4jReasoner			JavaBDDReasoner			JaCoPReasoner		
	Mutants	Alive	Score	Mutants	Alive	Score	Mutants	Alive	Score
VoidFM	55	20	63.6	75	12	84.0	8	0	100
ValidProduct	109	4	96.3	129	7	94.6	61	0	100
Products	86	1	98.8	130	2	98.5	37	0	100
#Products	57	1	98.2	77	2	97.4	13	0	100
Variability	82	1	98.8	104	2	98.1	36	0	100
Commonality	109	1	99.1	131	2	98.5	66	0	100
DeadFeatures	-	-	-	-	-	-	80	0	100
Total	**498**	**28**	**94.4**	**646**	**27**	**95.8**	**301**	**0**	**100**

Table 6.5: *Mutants execution results.*

6.3.1.3 Refinement

As shown in previous sections, mutation testing is an effective means to measure the effectiveness of a test suite. However, information provided by mutation testing can also be used to guide the creation of new test cases that kill the remaining alive mutants and strengthen the final test suite [157]. Following this approach, we designed a number of test cases to kill remaining undetected mutants until obtaining a score of 100% in the three FaMa reasoners. A total of 27 new test cases were created and executed. For instance, alive mutants guided us to the creation of a couple test cases to ensure that alternative relationships are not processed as or-relationships and vice–versa (e.g. test cases *VM-21* and *VM-22* in [149]). Out of the 27 test cases created, we selected those test cases that showed to be effective in killing mutants in at least two of the three subject reasoners and added them to our suite. As a result, 10 test cases were added to the initial test suite increasing the number of these until 190.

6.3.2 Evaluation using real faults

For a further evaluation of our approach, we checked the effectiveness of our tool in detecting real faults. In particular, we first studied a motivating fault found in the literature. Then, we used our test suite to test the release 1.0 alpha of the FaMa framework, detecting one defect. These results are next reported.

6.3.2.1 Motivating fault found in the literature

Consider the work of Batory in SPLC'05 [11], one of the seminal papers in the community of automated analysis of feature models. The paper included a bug (later fixed[†2]) in the mapping of a feature model to a propositional formula. We implemented this wrong mapping into a mock reasoner for FaMa and checked the effectiveness of our approach in detecting the fault.

Figure §6.6 illustrates an example of the wrong output caused by the fault. This manifests

[†2]ftp://ftp.cs.utexas.edu/pub/predator/splc05.pdf

itself in alternative relationships whose parent feature is not mandatory making reasoners to consider as valid product those including multiple alternative subfeatures and excluding the parent feature (P3). As a result, the set of products returned by the tool is erroneously larger than the actual one. For instance, the number of products returned by our faulty tool when using the model in Figure §2.6 as input is 3,584 (instead of the actual 2,016). Note that this is a motivating fault since it can easily remain undetected even when using an input with the problematic pattern. Hence, in the previous example (either with *"security"* feature as mandatory or optional), the mock tool correctly identifies the model as non void (i.e. it represents at least one product), and so the fault remains latent.

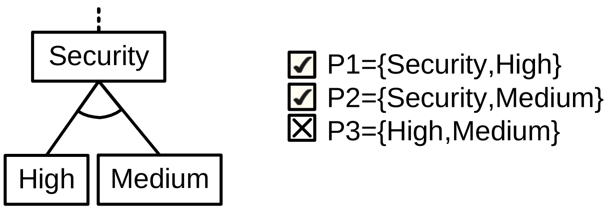


Figure 6.6: *Wrong set of products obtained with the faulty reasoner.*

We implemented our test cases using JUnit and tested our faulty tool. The fault was detected by our test suite in the operations *VoidFM*, *Product*, *#Products*, *Variability* and *Commonality* remaining latent in the operations *ValidProduct* and *DeadFeatures*. These two operations required a very specific pattern to reveal the fault not included in the inputs of our test suite. This gives an idea of the complexity of testing in this domain. We found this fault sufficiently motivating to extend our suite with test cases that detect it in all the operations. Thus, we designed two new test cases to detect the fault in the operation *ValidProduct* and *DeadFeatures* and added them to our test suite resulting in a total of 192 test cases.

6.3.2.2 FaMa Framework

Finally, we evaluated our tool by trying to detect faults in *FaMa Framework v1.0 alpha*. We executed the 192 test cases of our refined test suite. Tests revealed one defect. The fault affected the operations *ValidProduct* and *Commonality* in Sat4jReasoner. The source of the problem was a bug in the creation of propositional clauses in the so-called staged configurations, a new feature of the tool.

6.4 Test suite summary and discussion

Table §6.6 summarizes the general aspects of the refined test suite using the common terms of the IEEE Standard for Software Testing Documentation [78]. For each operation, the number of test cases and the testing techniques used are presented (EP: Equivalence Partitioning, PT: Pairwise Testing, BVA: Boundary-Value Analysis, EG: Error Guessing). Global inputs constraints specify constraints that must be true for every input in the set of associated test cases. For the sake of simplicity, two main input constraints were imposed, namely: i) input feature models must be syntactically correct (e.g. checking models for conformance to a metamodel), and ii) all input parameters of the analysis operations must be provided. An operation is said to pass the test (so-called pass criteria) when all the test cases associated to that operation are successful.

Test suite identifier: FaMa Test Suite v1.2		
Operations tested	**Test cases**	**Techniques used**
Void FM	24	EP, PT
Valid product	63	EP, PT, BVA
Products	21	EP, PT
Number of products	21	EP, PT
Variability	21	EP, PT
Commonality	33	EP, PT, BVA
Dead features	9	EG
Total	**192**	
Global input constraints:		
- Input FMs must be syntactically correct		
- All input parameters are required		
Operation pass criteria:		
- Pass 100% of associated test cases		

Table 6.6: *General overview of the FaMa Test Suite.*

Trying to be exhaustive when testing feature model analyses tools can easily increase the number of test cases to an unmanageable level. To keep a reasonable balance between number of test cases and test coverage, we kept in mind a number of generic recommendations from the testing literature, namely: i) we gave priority to those decisions reducing the number of test cases, ii) we avoided redundancies by designing each test case to reveal a single type of fault, and iii) we designed simple test cases whose output could be worked out manually to avoid test themselves to become error-prone.

We remark that the potential users of the suite are every tool supporting the analysis of feature model. This can be automated by simply implementing the test cases in the desired platform and executing them. A complete list of the test cases that compose the suite is reported in [149]. To facilitate its implementation, input models used in the test cases are also available in XML format in the FaMa Web site [58].

6.5 Summary

In this chapter, we have presented a set of implementation–independent test cases to validate the functionality of tools supporting the analysis of feature models. Through the implementation of our test cases, faults can be rapidly detected assisting in the development of feature model analysis tools and improving their reliability and quality. These can be used either in isolation or as a suitable complement for further testing methods such as white–box testing techniques or automated test data generators. For its design, we used popular techniques from the software testing community to assist us on the creation of a representative set of input–output combinations. To evaluate its effectiveness, we applied mutation testing on three open source feature model analysis tools integrated into the FaMa framework. These tools use different underlying paradigms and were coded by different developers what provides the necessary heterogeneity for the evaluation. We initially obtained an average mutation score of 96.1% and refined our suite progressively until getting 100%. Once refined, our suite also showed to be effective in detecting real faults found in the literature and in a recent release of FaMa. Both, the test suite documentation and the inputs models used in the test cases are ready–to–use and available at the FaMa Web Site. We intend this contribution to be a first effort toward the development of a widely accepted test suite to support functional testing in the community of automated analysis of feature models.

The results described in this chapter were presented in the 5th Software Product Lines Testing Workshop [143] and the IET Software journal [150].

Chapter 7

A test data generator for the analysis of feature models

The function of good software is to make the complex appear to be simple.

Grady Booch, 1955–
American software engineer and scientist

In the previous chapter, we presented a manually–designed test suite for functional testing on the analysis of feature models. Although effective, we found several practical limitations in our approach. One of the main limitations was the difficulty to calculate the expected output of the test cases which we found to be time-consuming, error-prone and in most cases infeasible due to the combinatorial complexity of the analyses, i.e. oracle problem. In this chapter, we propose using metamorphic testing to automate the generation of test data for feature model analysis tools overcoming the oracle problem. An automated test data generator is presented and evaluated to show the feasibility of our approach. Using this generator, complex feature models representing millions of products can be efficiently generated and used to test a number of analyses on the feature models automatically. Our evaluation results using mutation testing and real faults reveal that most faults can be automatically detected within a few seconds.

In Section §7.1, we introduce our contribution. A detailed description of our metamorphic relations and test data generator is presented in Section §7.2. Section §7.3 describes the evaluation of our approach in different scenarios as well as the comparison with FaMa Test Suite. We show how our approach can be refined by combining it with other test case selection strategies in Section §7.4. Section §7.5 discusses the main threats to validity of our work. In Section §7.6, we present the related works in the field of metamorphic testing and compare them with our approach. Finally, we summarize the chapter in Section §7.7.

7.1 Introduction

In the previous chapter, we gave a first step to address the problem of functional testing on the analyses of feature models. In particular, we presented a set of manually designed test cases, the so-called FaMa Test Suite (FaMa TeS), to validate the implementation of the analyses on feature models. Although effective, we found several limitations in our manual approach that motivated this work. First, the number of test cases is very high despite the fact that we did our best to keep it low during the design. Current version of the suite contain 192 test cases addressing 7 different analysis operations. This means that a suite that would cover the 30 analysis operations reported in the literature could easily have thousands of test cases hindering the manipulation and maintenance of the suite. Second, the suite is not generic, that is, specific test cases must be designed for each operation which is tedious and time-consuming. Third, evaluation results with artificial and real faults showed room for improvement in terms of efficacy. Finally, the manual design of new test cases relied on the ability of the tester to decide whether the output of an analysis was correct. We found this was time–consuming, error–prone and in most cases infeasible due to the combinatorial complexity of the analyses. As a result, we were force to use small and in most cases oversimplistic input models whose output could be calculated by hand. This limitation, also found in many other software testing domains, is known as the oracle problem [189] i.e. impossibility to determine the correctness of a test output.

In this chapter, we propose using metamorphic testing for the automated generation of test data for the analyses of feature models. In particular, we present a set of metamorphic relations between feature models and their set of products and a test data generator based on them. Given a feature model and its known set of products, our tool generates a set of neighbouring models together with their associated sets of products. Complex feature models representing millions of products can be efficiently generated by applying this process iteratively. Once generated, products are automatically inspected to get the expected output of a number of analyses over the models. Key benefits of our approach are that it removes the oracle problem and is highly generic being suitable to test any operation extracting information from the set of products of a feature model. In order to show the feasibility of our approach, we evaluated the ability of our test data generator to detect faults in three main scenarios. First, we introduced hundreds of artificial faults (i.e. mutants) into three of the analysis reasoners integrated into the FaMa framework and checked the effectiveness of our generator to detect them. As a result, our automated test data generator found more than 98.5% of the faults in the three reasoners with average detection times under 7.5 seconds. Second, we developed a mock tool including a motivating fault found in the literature and checked the ability of our approach to detect it automatically. As a result, the fault was detected in all the operations tested with a score of 91.4% and an average detection time of 23.5 seconds. Finally, we evaluated our approach with recent releases of two real tools for the analysis of feature models, FaMa and SPLOT, detecting two defects in each of them.

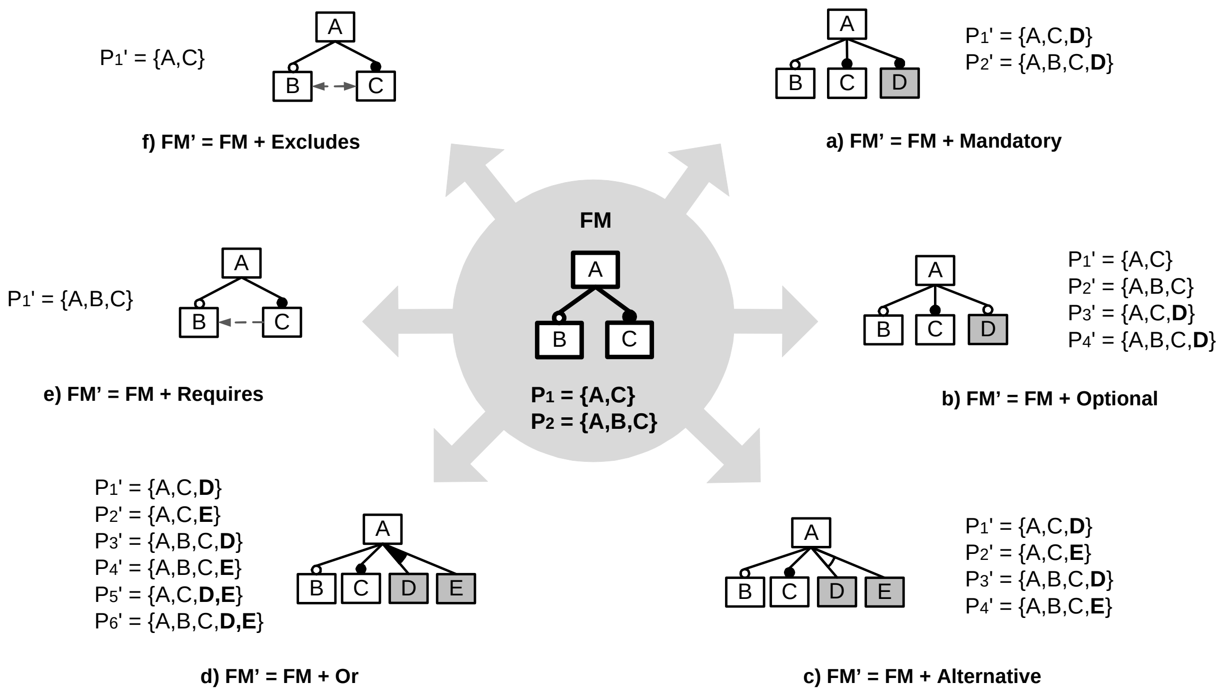


Figure 7.1: *Some examples of neighbour feature models.*

7.2 Automated metamorphic testing on the analysis of feature models

In this section, we detail our proposal. We first present a set of metamorphic relations to relate feature models and the set of products that these represent. Then, we show how these relations can be used for the automated generation of test data for the analysis of feature models. Finally, we present the implementation of our approach.

7.2.1 Metamorphic relations on feature models

In this section, we define a set of metamorphic relations between feature models (i.e. input) and their corresponding set of products (i.e. output). These metamorphic relations are derived from the basic modelling elements found in feature models, that is, the different types of relationships and cross–tree constraints among features. In particular, we relate feature models using the concept of neighbourhood. Given a feature model, FM, we say that FM' is a *neighbour model* if it can be derived from FM by adding or removing a relationship or constraint R. The metamorphic relations between the products of a model and the one of their neighbours are then determined by R as follows:

Mandatory. Consider the neighbours models and associated set of products depicted in Figure §7.1. FM' in Figure §7.1(a) is created from FM by adding a mandatory feature (D) to it, i.e. they are neighbours. The semantics of mandatory relationships state that mandatory features must always be part of the products in which is parent feature appears. Based on this, we conclude that the set of expected products of *FM'* is incorrect if it does not preserve the set of products of *FM* extending it by adding the new mandatory feature (D) in all the products including its parent feature (A). In the example, therefore, this relation is fulfilled. Formally, let f be the mandatory feature added to the model and pf its parent feature, D and A in the example respectively. Consider the functions $products(FM)$, returning the set of products of an input feature models, and $features(P)$, returning the set of features of a given product. We use the symbol '#' to refer to the cardinality (i.e. number of elements) of a set. We define the relation between the set of products of FM and the one of FM' as follows:

$$\begin{aligned}
&\#products(FM') = \#products(FM) \wedge \\
&\forall P'(P' \in products(FM') \Leftrightarrow \exists P \in products(FM) \cdot \\
&(pf \in features(P) \wedge P' = P \cup \{f\}) \vee \\
&(pf \notin features(P) \wedge P' = P))
\end{aligned}$$

Optional. Let f be the optional feature added to the model and pf its parent feature. An example is presented in Figure §7.1(b) with f = D and pf = A. Consider the function $filter(FM, S, E)$ that returns the set of products of FM including the features of S and excluding the features of E. The metamorphic relation between the set of products of FM and that of FM' is defined as follows:

$$\begin{aligned}
&\#products(FM') = \#products(FM) + \#filter(FM, \{pf\}, \emptyset) \wedge \\
&\forall P'(P' \in products(FM') \Leftrightarrow \exists P \in products(FM) \cdot \\
&P' = P \vee (pf \in features(P) \wedge P' = P \cup \{f\}))
\end{aligned}$$

Alternative. Let C be the set of alternative subfeatures added to the model and pf their parent feature. In Figure §7.1(c), C={D, E} and pf = A. The relation between the set of products of FM and FM' is defined as follows:

$$\begin{aligned}
&\#products(FM') = \#products(FM) + (\#C - 1)\#filter(FM, \{pf\}, \emptyset) \wedge \\
&\forall P'(P' \in products(FM') \Leftrightarrow \exists P \in products(FM) \cdot \\
&(pf \in features(P) \wedge \exists c \in C \cdot P' = P \cup \{c\}) \vee \\
&(pf \notin features(P) \wedge P' = P))
\end{aligned}$$

Or. Let C be the set of subfeatures added to the model and pf their parent feature. For instance, in Figure §7.1(d), C={D, E} and pf = A. We denote with $\mathcal{P}_I(C)$ the powerset of C (the set of all subsets in C) excluding the empty set. This metamorphic relation is defined as follows:

$$\begin{aligned}
&\#products(FM') = \#products(FM) + (2^{\#C} - 2)\#filter(FM, \{pf\}, \emptyset) \wedge \\
&\forall P'(P' \in products(FM') \Leftrightarrow \exists P \in products(FM) \cdot \\
&(pf \in features(P) \wedge \exists S \in \mathcal{P}_I(C) \cdot P' = P \cup S) \vee \\
&(pf \notin features(P) \wedge P' = P)))
\end{aligned}$$

Requires. Let f and g be the origin and destination features of the new requires constraint added to the model. In Figure §7.1(e), $f = \mathsf{C}$ and $g = \mathsf{B}$. The relation between the set of products of FM and FM' is defined as follows:

$$\mathtt{products}(FM') = \mathtt{products}(FM) \setminus \mathtt{filter}(FM, \{f\}, \{g\})$$

Excludes. Let f and g be the origin and destination features of the new excludes constraint added to the model. This is illustrated in Figure §7.1(f) with $f = \mathsf{B}$ and $g = \mathsf{C}$. This metamorphic relation is defined as follows:

$$\mathtt{products}(FM') = \mathtt{products}(FM) \setminus \mathtt{filter}(FM, \{f, g\}, \emptyset)$$

7.2.2 Automated test data generation

The semantics of a feature model is defined by the set of products that it represents [140]. Most analysis operations on feature models can be answered by inspecting this set adequately. Based on this, we propose a two–step process to automatically generate test data for the analyses of feature models as follows:

Feature model generation. We propose using previous metamorphic relations together with model transformations to generate feature models and their respective set of products. Note that this is a singular application of metamorphic testing. Instead of using metamorphic relations to check the output of different computations, we use them to actually compute the output of follow–up test cases. Figure §7.2 illustrates an example of our approach. The process starts with an input feature model whose set of products is known. A number of step–wise transformations are then applied to the model. Each transformation produces a neighbour model as well as its corresponding set of products according to the metamorphic relations. Transformations can be applied either randomly or using heuristics. This process is repeated until a feature model (and corresponding set of products) with the desired properties (e.g. number of features) is generated.

Test data extraction. Once a feature model with the desired properties is created, it is used as non-trivial input for the analysis. Similarly, its set of products is automatically inspected to get the output of a number of analysis operations i.e. any operation that extracts information from the set of products of the model. As an example, consider the model and set of products generated in Figure §7.2 and the analysis operations listed in Table §7.1 (fully described in Section §3.2). We can obtain the expected output of all of them by simply answering the following questions:

- *Is the model void?* No, the set of products is not empty.
- *Is P=*{$\mathsf{A}, \mathsf{C}, \mathsf{F}$}* a valid product?* Yes. It is included in the set.
- *How many different products represent the model?* 6 different products.

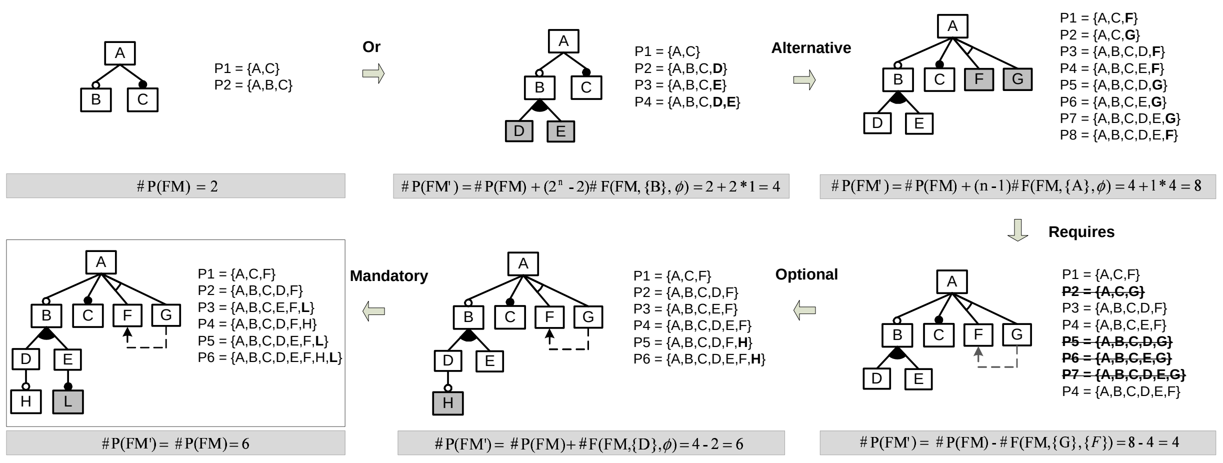


Figure 7.2: *An example of random feature model generation using metamorphic relations.*

- *What is the variability of the model?* $6/(2^9 - 1) = 0.011$
- *What is the commonality of feature* B*?* Feature B is included in 5 out of the 6 products of the set. Therefore its commonality is 83.3%
- *Does the model contain any dead feature?* Yes. Feature G is dead since it is not included in any of the products represented by the model.

Operation	**Description**
VoidFM	Informs whether the input feature model is void or not
ValidProduct	Informs whether the input product belongs to the set of products of a given model
Products	Returns the set of products of a feature model
#Products	Returns the number of products represented by a feature model
Variability	Returns the variability degree of a feature model
Commonality	Returns the percentage of products in which a given feature appears

Table 7.1: *Analysis operations tested.*

We may remark that we could have also used a 'pure' metamorphic approach, start with a known feature model, transform this to obtain a neighbour model, and use metamorphic relations to check the outputs of the tool under test. However, this strategy would require to define metamorphic relations for each operation. In contrast, we propose to use the metamorphic relations to compute the output of follow-up test cases instead of simply comparing the results of different tests. Starting from a trivial test case, we can generate increasingly larger and more complex test cases making sure that the metamorphic relations are fulfilled at each step. This allows us to define the metamorphic relations for a single operation, *Products*, from which we derive the expected output of many of the other analyses on feature models. A key benefit of our approach is that it can be easily automated enabling the generation and execution of test cases without the need for a human oracle.

Finally, we would like to emphasize that the operations presented are only some examples of the analyses that can be tested using our approach. We estimate that this technique could be used to test, at least, 16 out of the 30 analysis operations identified in [21]. The operations out of the scope of our approach are mainly those looking for specific patterns in the feature tree.

7.2.3 A test data generator

As a part of our proposal, we implemented a test data generator relying on our metamorphic relations. The tool receives a feature model and its associated set of products as input and returns a modified version of the model and its expected set of products as output. If no inputs are specified, a new model is generated from scratch.

Our tool applies random transformations to the input model increasing its size progressively. The set of products is efficiently computed after each transformation according to the metamorphic relations presented in Section §7.2.1. Transformations are performed according to a number of parameters including number of features, percentage of constraints, maximum branching factor and percentage of each type of relationship to be generated.

The number of products of a feature model increases exponentially with the number of features. This was a challenge during the development of our tool causing frequent time deadlocks and memory overflows. To overcome these problems, we optimized our implementation using efficient data structures (e.g. boolean arrays) and limited the number of products of the models generated. Using this setup, feature models with up to 11 million products were generated in a standard laptop machine within a few seconds.

Our test data generator has been integrated into our framework BeTTy (see Appendix §A). This system provides a number of capabilities for benchmarking and testing in the context of feature models including test data generators as well as readers and writers for different formats. Figure §7.3 depicts a random feature model generated with our test data generator and exported from BeTTy to the graph visualization tool GraphViz [69]. The model has 20 features and 20% of constraints. Its set of products contains 22,832 different feature combinations.

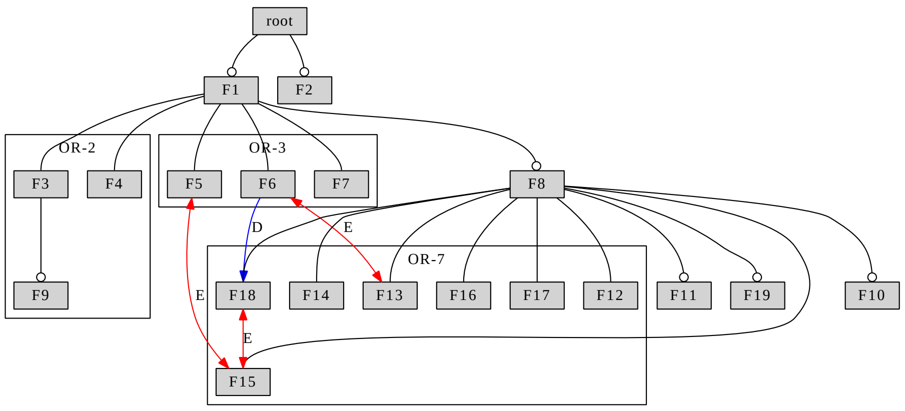


Figure 7.3: *Sample input feature model generated with our tool.*

7.3 Evaluation

7.3.1 Evaluation using mutation testing

In order to measure the effectiveness of our proposal, we evaluated the ability of our test data generator to detect faults in the software under test (i.e. so–called fault-based adequacy criterion). To that purpose, we applied mutation testing on the FaMa Framework in a similar way as we did to evaluate our manual suite. The setup used in our experiments and the analysis of the results are next presented.

7.3.1.1 Experimental setup

As with our manual suite, we selected the FaMa Framework as a good candidate to be mutated. In particular, we selected three of the reasoners integrated into the framework, namely: Sat4jReasoner v0.9.2 (using satisfiability problems by means of Sat4j solver [138]), JavaBDDReasoner v0.9.2 (using binary decision diagrams by means of JavaBDD solver [80]) and JaCoPReasoner v0.8.3 (using constraint programming by means of JaCoP solver [79]). Each one of these reasoners uses a different paradigm to perform the analyses and was coded by different developers, providing the required heterogeneity for the evaluation of our approach. For each reasoner, the analysis operations listed in Table §7.1 were tested. The operation *DeadFeatures*, however, was tested in JaCoPReasoner exclusively since it was the only reasoner implementing it.

Test cases were generated randomly using our test data generator as described in Section §7.2.2. In the cases of operations receiving additional inputs apart from the feature model (e.g. valid product), the additional inputs were selected using a basic partition equivalence strategy. For each operation, test cases with the desired properties were generated and run until a fault was found or a timeout was exceeded. Feature models were generated with an initial size of 10 features and 10% (with respect to the number of features) of constraints for efficiency. This size was then incremented progressively according to a configurable increasing factor. This factor was typically set to 10% and 1% (every 20 test cases generated) for features and constraints respectively. The maximum size of the set of products was equally limited for efficiency. This was configured according to the complexity of each operation and the performance of each reasoner with typical values of 2000, 5000 and 11000000.

The mutation tool and configuration used in our evaluation was identical to the used for the evaluation of our test suite, described in Section §6.3.1.1. In particular, we followed three steps for the evaluation of our approach, namely:

i. *Reasoners testing.* Prior to their analysis, we checked whether the original reasoner passed all the tests. A timeout of 60 seconds was used. As a result, we detected and fixed a defect affecting the computation of the set of products in JaCoPReasoner. We found this fault to be especially motivating since it was also present in the current release of FaMa (see Section §7.3.2.2 for details).

ii. *Mutants generation.* We applied all the traditional mutation operators available in MuClipse, a total of 15. Specific mutation operators for object–oriented code were discarded to keep the number of mutants manageable. For details about these operators we refer the reader to [95].

iii. *Mutants execution.* For each mutant, we ran our test data generator and tried to find a test case that kills it. An initial timeout of 60 seconds was set for each execution. This timeout was then repeatedly incremented by 60 seconds (until a maximum of 600) with remaining alive mutants recorded. Equivalent mutants were manually identified and discarded after each execution.

Both the generation and execution of mutants was performed in a laptop machine equipped with an Intel Pentium Dual CPU T2370@1.73GHz and 2048 MB of RAM memory running Windows Vista Business Edition and Java 1.6.0_05.

7.3.1.2 Analysis of results

Table §7.2 recalls the information about the size of the reasoners and the number of generated mutants. Lines of code (LoC) do not include blank lines and comments. Out of the 760 generated mutants, 103 of them (i.e. 13.5%) were identified as semantically equivalent. In addition to these, we manually discarded 87 mutants (i.e. 11.4%) affecting secondary functionality of the subject programs (e.g. computation of statistics) not addressed by our current test data generator.

Reasoner	LoC	Mutants	Equivalent	Discarded
Sat4jReasoner	743	262	27	47
JavaBDDReasoner	625	302	28	37
JaCoPReasoner	791	196	48	3
Total	**2,159**	**760**	**103**	**87**

Table 7.2: *Mutants generation results.*

Tables §7.3, §7.4 and §7.5 show the results of the mutation process on Sat4jReasoner, JavaBDDReasoner and JaCoPReasoner respectively. For each operation, the number of classes involved, number of executed mutants, test data generation results and mutation score are presented. Test data results include average and maximum time required to kill each mutant, average and maximum number of test cases generated to kill a mutant and maximum timeout that showed to be effective in killing any mutant, i.e. further increments in the timeout (until the maximum of 600s) did not kill any new mutant.

Note that the functionality of each operation was scattered in several classes. Some of these were used in more than one operation. Mutants on these reusable classes were evaluated separately with the test data of each operation using them for more accurate mutation scores. This explains why the number of executed mutants on each reasoner (detailed in Tables §7.3, §7.4 and §7.5) is higher than the number of mutants generated for that reasoner (showed in Table §7.2).

Results revealed an overall mutation score of over 98.5% in the three reasoners. Operations *Products*, *#Products*, *Variability* and *Commonality* showed a mutation score of 100% in all the

Operations		Executed Mutants		Test Data Generation					Score
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	**Timeout (s)**	
VoidFM	2	55	0	37.6	566.5	95.1	414	600	100
ValidProduct	5	109	3	4.3	88.6	12	305	120	97.2
Products	2	86	0	0.6	3.4	1.5	12	60	100
#Products	2	57	0	0.7	2.4	1.8	8	60	100
Variability	3	82	0	0.6	1.7	1.3	5	60	100
Commonality	5	109	0	0.6	3.8	1.5	13	60	100
Total	**19**	**498**	**3**	**7.4**	**566.5**	**18.9**	**414**		**99.4**

Table 7.3: *Test data generation results in Sat4jReasoner.*

Operations		Executed Mutants		Test Data Generation					Score
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	**Timeout (s)**	
VoidFM	2	75	3	6.6	111.7	29.3	350	120	96
ValidProduct	5	129	5	1	34.6	3.8	207	60	96.1
Products	2	130	0	0.7	34.6	1.4	12	60	100
#Products	2	77	0	0.5	1.4	1.6	6	60	100
Variability	3	104	0	0.5	2.4	1.6	12	60	100
Commonality	5	131	0	0.5	3	1.5	16	60	100
Total	**19**	**646**	**8**	**1.6**	**111.7**	**6.5**	**350**		**98.7**

Table 7.4: *Test data generation results in JavaBDDReasoner.*

Operations		Executed Mutants		Test Data Generation					Score
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	**Timeout (s)**	
VoidFM	2	8	0	1.5	8.3	11.3	83	60	100
ValidProduct	5	61	0	0.7	1.2	1.3	5	60	100
Products	2	37	0	0.5	0.7	1	1	60	100
#Products	2	13	0	0.5	0.7	1	1	60	100
Variability	3	36	0	0.5	0.7	1	1	60	100
Commonality	5	66	0	0.5	0.7	1.1	3	60	100
DeadFeatures	5	80	0	0.8	2.1	2.3	14	60	100
Total	**24**	**301**	**0**	**0.7**	**8.3**	**2.7**	**83**		**100**

Table 7.5: *Test data generation results in JaCoPReasoner.*

reasoners with an average number of test cases required to kill each mutant under 2. Similarly, the operation *DeadFeatures* revealed a mutation score of 100% in JaCoPReasoner with an average number of test cases of 2.3. This suggests that faults in these operations are easily killable. On the other hand, faults in the operations *VoidFM* and *ValidProduct* appeared to be more difficult to detect. We found that mutants on these operations required input models to have a very specific pattern in order to be revealed. As a consequence of this, the average time and number of test cases required for these operations were noticeable higher than for the other analysis operations tested.

The maximum average time to kill a mutant was 7.4 seconds. In the worst case, our test data generator spent 566.5 seconds before finding a test case that killed the mutant. In this time, 414 different test cases were generated and run. This shows the efficiency of the generation process. The maximum timeouts required to kill a mutant were 600 seconds for the operation *VoidFM*, 120 for the operation *ValidProduct* and 60 seconds for the rest of analysis operations. This gives an idea of the minimum timeout that should be used when applying our approach in other scenarios.

Figure §7.4 depicts a spread graph with the size (number of features and constraints) of the feature models that killed mutants in the operation *VoidFM*. As illustrated, small feature models were in most cases sufficient to find faults. This was also the trend in the rest of the operations. This means that feature models with an initial size of 10 features and 10% of cross-tree constraints were complex enough to exercise most of the features of the analysis reasoners under test. This suggests that the procedure used for the generation of models, starting from smaller and moving progressively to bigger ones, is adequate and efficient.

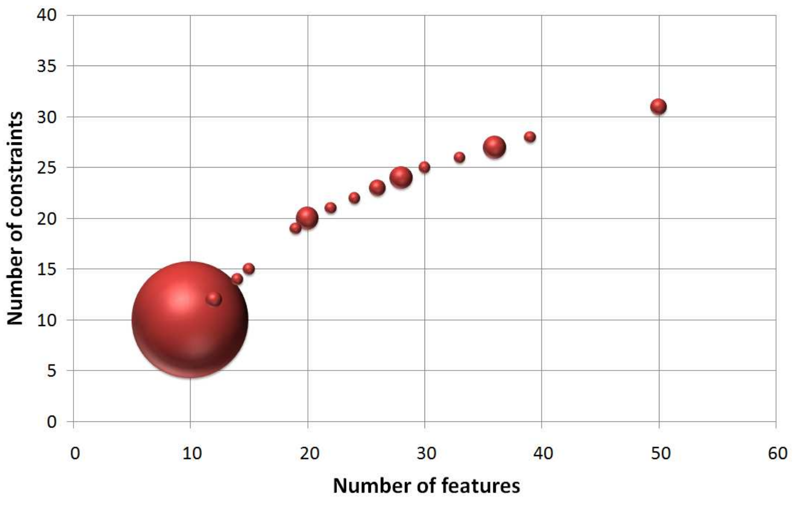


Figure 7.4: *Size of the feature models killing mutants in the operation* VoidFM*.*

Finally, we may mention that experimentation with Sat4jReasoner revealed a serious defect affecting its scalability. The reasoner created a temporary file for each execution but it did not delete it afterward. We found that the more temporary files were created, the slower became the creation of new ones with delays of up to 30 seconds in the executions of operations. Once detected, the defect was fixed and the experiments repeated. This suggests that our approach could also be applicable to scalability testing.

For more details about the evaluation of our test data generator using mutation testing on FaMa, we refer the reader to Appendix §C. The content of the appendix is based on an experience report presented by the authors in the Information and Software Technology special issue on Mutation Testing [152].

7.3.2 Evaluation using real faults

In this section, we present the evaluation results of our test data generator with some real faults found in the literature and two tools for the analysis of feature models.

7.3.2.1 A motivating fault

In this section, we report on the results obtained when using our test data generator to detect the motivating fault reported by Batory in SPLC'05 [11]. This fault was fully described in the previous chapter (Section §6.3.2.1). In particular, we manually inserted the bug into a mock reasoner for FaMa using the CSP-based solver Choco [44] and checked the effectiveness of our approach in detecting it.

Table §7.6 depicts the results of the evaluation. The testing procedure was similar to the one used with mutation testing. A maximum timeout of 600 seconds was used. The results are based on 10 executions. The fault was detected in all the executions performed in 6 out of 7 operations. Most of the average and maximum times were higher than the ones obtained when using mutants but still low being 191.9 seconds (3.2 minutes) in the worst case. The fault remained latent in 40% of the executions performed in the *ValidProduct* operation. When examining the data, we concluded that this was due to the basic strategies used for the selection of inputs products for this operation. We presume that using more complex heuristic for this purpose would improve the results.

Operation	Av Time (s)	Max Time (s)	Av TCs	Max TCs	Score
VoidFM	101.2	191.9	294.6	366	100
ValidProduct	41.6	91.8	146.8	312	40
Products	1.8	4.6	4.5	14	100
#Products	2.9	7.9	9.0	28	100
Variability	2.2	3.2	6.1	10	100
Commonality	2.1	4.8	5.6	15	100
DeadFeatures	12.8	29.2	42.3	101	100
Total	**23.5**	**191.9**	**72.7**	**366**	**91.4**

Table 7.6: *Evaluation results using a motivating fault reported in the literature.*

7.3.2.2 FaMa Framework

We also evaluated our tool by trying to detect faults in *FaMa Framework v1.0 alpha*. A timeout of 600 seconds was used for all the operations since we did not know *a priori* the existence of faults. For each operation, we ran our test data generator 10 times. Tests revealed two defects in all the executions (see Table §7.7). The first one, also detected during our experimental work with mutation, was caused by an unexpected behaviour of JaCoP solver when dealing with certain heuristics and void models in the operation *Products*. In these cases, the solver did not instantiate an array of variables raising a null pointer exception. This fault was detected in 142.9 seconds on average. The second fault, detected in less than one second in all executions, affected the operations *ValidProduct* and *Commonality* in Sat4jReasoner. The source of the problem was a bug in the creation of propositional clauses in the so-called staged configurations, a new feature of the tool. Both bugs were fixed in the new version of the tool.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score**
JaCoP-Products	142.9	198.6	437.3	605	100
Sat4j-ValidProduct	0.6	0.7	1	1	100
Sat4j-Commonality	0.6	0.6	1	1	100
Total	**48**	**198.6**	**146.4**	**605**	**100**

Table 7.7: *Evaluation results with FaMa.*

7.3.2.3 SPLOT

Software Product Lines On-line Tools (SPLOT) [107, 159] is a Web portal providing a complete set of tools for on-line editing, analysis and storage of feature models. It supports a number of analyses on cardinality-based feature models using propositional logic by means of the Sat4j and JavaBDD solvers. The authors of SPLOT kindly sent us a standalone version[†1] of their system to evaluate our automated test data generator. In particular, we tested the operations *VoidFM*, *#Products* and *DeadFeatures* in SPLOT. As with FaMa, we used a timeout of 600 seconds and tested each operation 10 times to get averages. Tests revealed two defects in all the executions (see Table §7.8). The first one, detected in less than one second on average, affected all operations on the SAT-based reasoner. With certain void models, the reasoner raised an exception (*org.sat4j.specs.ContradictionException*) and no result was returned. The second bug, detected in about 0.5 seconds in all cases, was related with cardinalities in the BDD-based tool. We found that the reasoner was not able to process cardinalities other than [1,1] and [1,*]. As a consequence of this, input models including or-relationships specified as [1,n] (n being the number of subfeatures) caused a failure in all the operations tested. Faults detected in the standalone version of the tool were also observed in the online version of SPLOT. We may remark that the authors confirmed the results and told us that they were aware of these limitations.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score**
Sat4j-VoidFM	0.7	1.3	26.7	66	100
Sat4j-#Products	1	2	26.1	66	100
Sat4j-DeadFeatures	0.9	2.2	38.3	134	100
JavaBDD-VoidFM	0.4	0.5	1.5	2	100
JavaBDD-#Products	0.4	0.5	1.9	5	100
Total	**0.7**	**2.2**	**18.9**	**134**	**100**

Table 7.8: *Evaluation results with SPLOT.*

[†1]SPLOT does not use a version naming system. We tested the tool as it was in February 2010.

7.3.3 Comparison with a manual test suite

In this section, we compare the ability to detect faults of our automated test data generator and the test suite (FaMa TeS) presented in the previous chapter. To that purpose, we compare the evaluation results obtained with both techniques. Notice that this is a fair comparison since both approaches were evaluated with exactly the same mutants and real faults. We may also remark that we refer to our manual suite as the set of 180 test cases presented in Section §6.2. The refined version of the suite, with 192 test cases, was discarded since it was specifically extended to detect the subject mutants and faults used for the comparison.

Table §7.9 recalls the results obtained when using FaMa TeS to kill the mutants in the FaMa reasoners. For each reasoner and operation, the total number of executed mutants, alive mutants and mutation score are presented. On the one hand, all mutants in JaCoPReasoner were killed by the manual suite equalling the results obtained with our metamorphic approach. On the other hand, mutation scores in Sat4jReasoner (94.4%) and JavaBDDReasoner (95.8%) were significantly lower than those obtained with our test data generator (99.4% and 98.7% respectively). This inferiority of the manual suite was also observed in the results of the evaluation with the bugs found in FaMa, SPLOT and the faulty reasoner (i.e. that including the motivating fault found in [11]). These results are depicted in Table §7.10. In the faulty reasoner, our automated test data generator detected the fault in all the operations meanwhile our manual suite failed to detect the defect in the operations *ValidProduct* and *DeadFeatures*. Similarly, the manual suite was unable to reveal the failure in the operation *Products* of JaCoPReasoner in FaMa 1.0.

Operation	Sat4jReasoner			JavaBDDReasoner			JaCoPReasoner		
	Mutants	Alive	Score	Mutants	Alive	Score	Mutants	Alive	Score
VoidFM	55	20	63.6	75	12	84.0	8	0	100
ValidProduct	109	4	96.3	129	7	94.6	61	0	100
Products	86	1	98.8	130	2	98.5	37	0	100
#Products	57	1	98.2	77	2	97.4	13	0	100
Variability	82	1	98.8	104	2	98.1	36	0	100
Commonality	109	1	99.1	131	2	98.5	66	0	100
DeadFeatures	-	-	-	-	-	-	80	0	100
Total	**498**	**28**	**94.4**	**646**	**27**	**95.8**	**301**	**0**	**100**

Table 7.9: *Mutants execution results of the manual test suite.*

From the results obtained and our experience working with FaMa TeS, we conclude that our automated metamorphic approach outperformed the manual suite in multiple ways. First, our automated generator was more effective than the manual suite, i.e. it detected more faults. Second, our metamorphic approach is highly generic so it can easily be adapted to test numerous analysis operation while the development of manual test cases is tedious and time-consuming. Also, manual test cases are trivially small while our current approach allows the efficient generation of large feature models representing million of products. Finally, and more important, our generator automatically checks the output of tests, removing the oracle problem found when using manual means. All these pieces of evidence support the effectiveness of our approach when compared to related testing mechanisms for feature model analysis tools in general, and

Fault	Automated Generator	Manual Test Suite
Faulty reasoner		
VoidFM	+	+
ValidProduct	+	-
Products	+	+
#Products	+	+
Variability	+	+
Commonality	+	+
DeadFeatures	+	-
Faults in FaMa and SPLOT		
FaMa-JaCoPProducts	+	-
FaMa-Sat4j	+	+
SPLOT-Sat4j	+	+
SPLOT-JavaBDD	+	+

Table 7.10: *Real faults detected by our test data generator and the manual suite.*

manual mechanisms in particular.

7.4 Refinement

In the approach presented previously, test cases are randomly generated from scratch for simplicity. However, it is known that metamorphic testing produces better results when combined with other test case selection strategies that generate the initial set of test cases [37, 38]. In this section, we propose refining our approach by using an initial set of input models that seed the generation of follow-up test cases. This initial set of models could guide the generator to search in specific error-prone areas improving the detection results. To show the feasibility of the proposal, we used the input models in our manual test suite (see Chapter §6 for details) as seed for the automated generation of test data. Later, we repeated the evaluation with mutants and real faults and checked how the input test cases had improved the efficiency and effectiveness of our automated generator.

As a preliminary step, we manually refined our suite by adding new test cases that kill the remaining alive mutants found during the evaluation with mutation. As mentioned previously, this is a natural step when using mutation to improve the quality of the test suite [157]. In order to avoid the suite being overfitted for the mutants under evaluation, we used the information provided by only one of the reasoners that was later excluded for the evaluation. In particular, we selected Sat4jReasoner since it was the one in which more mutants remained alive and therefore the one providing more feedback to improve our suite (see Table §7.9). As a result, 13 new test cases were added to the manual suite (from 180 to 193), i.e. those that killed the remaining alive mutants in Sat4jReasoner.

Figure §7.5 illustrates the steps we followed to use the input models of the refined manual suite to guide the generation of follow–up test cases. For each operation, the input models used

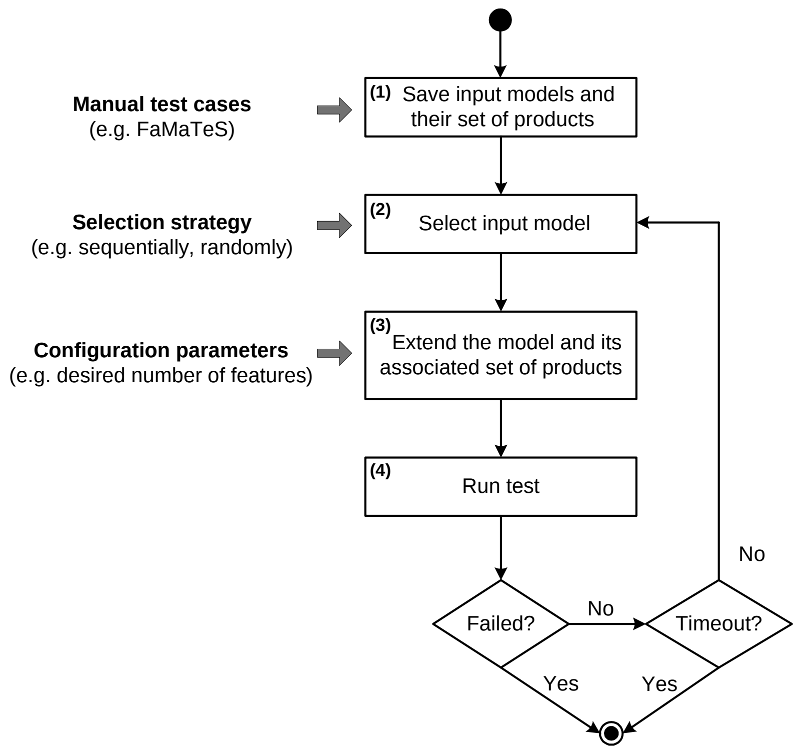

Manual test cases
(e.g. FaMaTeS)
Selection strategy
(e.g. sequentially, randomly)
Configuration parameters
(e.g. desired number of features)
(1) Save input models and their set of products
(2) Select input model
(3) Extend the model and its associated set of products
(4) Run test
Failed?
No
Timeout?
No
Yes
Yes

Figure 7.5: *Algorithm for the generation of test cases using a starting manual test suite.*

in their associated test cases in FaMa TeS and their corresponding set of products (calculated manually) are saved (step 1). Then, for each test case to be generated, a feature model is selected (step 2) and extended (step 3) by applying a set of step-wise random transformations to it. Each transformation produces a neighbour model as well as its corresponding set of products according to the metamorphic relations presented in Section §7.2.1. Once a feature model with the desired properties has been generated, the test case is run (step 4) and the execution stopped if a failure is revealed. Otherwise, a new input model from FaMa TeS is selected and the previous process repeated. In our current approach, initial input models are selected sequentially, however, other strategies (e.g. random selection) would also be feasible. A maximum timeout of 600 seconds was used for all the executions. The configuration parameters for the generation (e.g. desired number of features, increasing size factor, etc.) were set to the same values described in Section §7.3.1.1.

Table §7.11 depicts the mutants execution results of our refined generator. For each reasoner, the average detection time, maximum detection time, average number of test cases generated and mutation scores are presented. The last row shows the average values in the form *x* / *y* where *x* is the value obtained when using our initial approach (i.e. test cases are created randomly from scratch) and *y* is the value obtained when using the refined version of our generator (i.e. input models from FaMa TeS are used to guide the generation of test cases). As illustrated, the experiments revealed a significant improvement in the detection times and number of test cases generated before killing a mutant. In JavaBDDReasoner, for instance, the average detection time was reduced by 43.7% (from 1.6 to 0.9 seconds) and the number of test cases was reduced by 63% (from 6.5 to 2.4 test cases). This improvement was especially significant in the maximum detection times reduced by 63.9% (from 111.7 to 40.3 seconds) in JavaBDDReasoner and 79.5% (from 8.3 to 1.7 seconds) in JaCoPReasoner. We may mention that we found some cases, those with lowest times, in which our refined generator was slightly slower than our original approach. As expected, this was caused by the overhead introduced in the new program when loading the initial test set from XML files. Finally, we also found a slight improvement in the mutation score of JavaBDDReasoner, from 98.7% to 98.9%.

Operation	**JavaBDDReasoner**				**JaCoPReasoner**			
	Av Time (s)	**Max Time (s)**	**Av TCs**	**Score**	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Score**
VoidFM	1.5	25.7	5.8	97.3	0.8	1.7	2.3	100
ValidProduct	0.9	7.2	2.3	96.1	0.8	1.2	1.3	100
Products	1.0	40.3	1.5	100	0.8	1.1	1.0	100
#Products	0.7	1.5	1.5	100	0.9	1.1	1.1	100
Variability	0.7	3.5	1.6	100	0.8	0.9	1.0	100
Commonality	0.6	2.9	1.4	100	0.8	1.2	1.1	100
DeadFeatures	-	-	-	-	0.8	1.1	1.1	100
Total	1.6 / **0.9**	111.7 / **40.3**	6.5 / **2.4**	98.7 / **98.9**	0.7 / **0.8**	8.3 / **1.7**	2.7 / **1.3**	100 / **100**

Table 7.11: *Mutants execution results of our refined automated test data generator.*

The evaluation results with real faults, shown in Table §7.12, were similar to those obtained with mutants. The average detection times, for instance, were reduced by 41.7% (from 23.5 to 13.7 seconds) in the faulty reasoner and by 43.9% (from 36.2 to 20.3 seconds) in the real faults founds in FaMa and SPLOT. Results in the operation *VoidFM* of our faulty reasoner were especially positive with a reduction in the average detection time of 93.6%, from 101.2 seconds (see Table §7.6) to 6.4. The mutation score in the operation *ValidProduct* showed no improvement. Again, we think this is due to the basic strategies used for the selection of input products for this operation. More complex heuristic for this purpose could certainly yield better results. Finally, we may mention that the results obtained in the operation *DeadFeatures* of the faulty reasoner were much worse that those found in our original approach with an average detection time increasing from 12.8 seconds (see Table §7.6) to 41.3. Interestingly, it seems that starting the generation with models that already had some dead features affected negatively the detection of the fault.

These results support the feasibility of combining our test data generator with other testing strategies that generate the initial set of models for a more effective search of faults. However, while the improvement in detection times was noticeable, we may remark that we did not obtain significant improvements in terms of efficacy. Therefore, we encourage researchers and practitioners following our approach to assess carefully the trade–off between the effort required to develop an initial set of test cases and the expected gains in efficiency.

7.5 Threats to validity

We briefly discuss the threats to validity of our work.

Fault	Av Time (s)	Av TCs	Score
Faulty reasoner			
VoidFM	6.4	22	100
ValidProduct	39.1	145.8	40
Products	2.0	4.7	100
#Products	2.3	5.2	100
Variability	2.0	4.4	100
Commonality	2.9	7.1	100
DeadFeatures	41.3	151.9	100
Total	23.5 / **13.7**	72.7 / **48.7**	91.4 / **91.4**
Faults in FaMa and SPLOT			
FaMa-JaCoPProducts	79.2	244.0	100
FaMa-Sat4j	1.0	1.2	100
SPLOT-Sat4j	0.5	8.7	100
SPLOT-JavaBDD	0.4	1.9	100
Total	36.2 / **20.3**	117.6 / **63.9**	100 / **100**

Table 7.12: *Evaluation results of our refined generator using real faults.*

- **Subject reasoners**. Our mutation results apply only to three of the reasoners integrated into the FaMa framework and therefore could not extrapolate to other programs. Nevertheless, we may remark that each one of these reasoners use a different technique to automate the analysis and were coded by different developers providing the required level of heterogeneity for our evaluation.
- **Equivalent mutants**. The detection of equivalent mutants, an undecidable problem in general, was performed by hand resulting in a tedious and error-prone task. Thus, we must concede a small margin of error in the data regarding equivalence. We remark, however, that results were taken from three different reasoners providing a fair confidence in the validity of the average data. Furthermore, equivalence results were also confirmed by the results obtained by our manual suite.
- **Real faults**. The number of real faults in our study was not large enough to allow us to draw general conclusions. However, we may emphasize that these were collected from both the literature and real tools providing a sufficient degree of representativeness. These faults were harder to detect than mutants in general and provided a good idea of the behaviour of our approach in real scenarios.

7.6 Related works on metamorphic testing

The related works in the field of metamorphic testing can be divided into three areas, namely:

Applications. Chen et al. [38] studied the application of metamorphic testing to address the oracle problem in numerical programs. A case study with partial differential equations was presented. Zhou et al. [200] presented several uses of metamorphic testing in the domains of graph theory, computer graphics, compilers and interactive software. Some metamorphic relations were proposed but no experimental results were reported. Later, in [39], the authors proposed a guideline for the selection of good metamorphic relations and presented two cases studies with the shortest path program and the critical path program. Experimental results of the evaluation of the metamorphic relations using manual mutation testing were reported. In [34], Chan et al. presented a metamorphic approach for integration testing in context–sensitive middleware–based applications. The authors identified functional relations that associate different execution sequences of a test case. Then, they used metamorphic testing to check the results of the test cases and find contradictions of those relations. Chan et al. [33] proposed an approach for online service testing and presented an experiment with a service-oriented calculator of arithmetic expressions to show the feasibility of their work. Chen et al. [36] proposed using metamorphic testing to test bioinformatic programs and presented experimental results with two of those programs.

Tools, frameworks and methods. Gotlieb and Botella [68] proposed an automated testing framework able to check metamorphic relations using constraint programming. Given a program and a metamorphic relation, their tool tries to find test data that violates the relation. Evaluation results with mutation testing were presented. Chan et al. [35] proposed a testing methodology for service-oriented applications based on metamorphic testing. The authors introduced the concept of metamorphic service. A *metamorphic service* is a service that calls the relevant services of the application and checks the metamorphic relations. Beydeda [26] proposed a method to enable self-testability of components using metamorphic testing. Murphy et al. [116] presented an extension to the Java Modeling Language (JML) and a tool able to process it. This extension allows users to specify metamorphic relations as annotations in the Java code. These annotations are later processed by their tool that generates test code that can be executed using JML runtime assertion checking, for ensuring that the specifications hold during program execution. Later, in [115], the authors presented a framework called Amsterdam to support metamorphic testing at the system level. They also presented an approach called *Heuristic Metamorphic Testing* to reduce false positives and address some cases of non-determinism. The authors extended their work in [114] presenting a new technique called *Metamorphic Runtime Checking*, a testing approach that automatically conducts metamorphic testing of individual functions during the program's execution. The authors also presented a framework called columbus and presented experimental results.

Integration of metamorphic testing with other testing techniques. Chen et al. [41] proposed a semi–proving method based on metamorphic testing and global symbolic evaluation. The proposed method verifies expected necessary properties for program correctness and identifies failure-causing inputs if such properties are not satisfied. Later, in [42], the authors presented an integrated method that combined metamorphic testing and fault–based testing by means of mutation testing. Chen et al. [40] proposed using metamorphic testing in combination with special values testing. Special test values are test values in which their expected results are well known and can be used to verify the program. Some examples with numerical programs were presented. Xie et al. [195] extend the spectrum–based fault localization method with metamorphic testing making it applicable to applications without a test oracle.

When compared to previous studies, our work contributes to the three main areas mentioned above as follows. First, we have presented the application of metamorphic testing to a novel domain, the analysis of feature models. In contrast to most related works, our metamorphic relations are derived from the operators of the models (i.e. types of relationships and constraints) rather than from the properties of the application domain in which they are used. Also, we have applied metamorphic testing in a slightly different way to that showed in related studies. In particular, we have used the metamorphic relations to compute the output of follow-up test cases instead of simply comparing the results of different tests. Starting from a trivial test case, we generate increasingly larger and more complex test data by making sure that the metamorphic relations are fulfilled at each step. This strategy allowed use to define the metamorphic relations for a single operation, *Products*, from which we derived the expected output of many of the other analyses on feature models. Second, we have presented a test data generator for the automated generation of test data based on our metamorphic relations. This generator is available as a part of the BeTTy framework. In contrast to related works, we have evaluated our test data generator using hundreds of automatically inserted mutants rather than manual mutation. We have also evaluated our approach with real faults found in the literature and current releases of several tools. We are not aware of any other study reporting the detection of real bugs using metamorphic testing. Finally, we have proposed a new integrated proposal combining our metamorphic approach and a black–box test suite showing experimental evidences of the gains obtained in terms of efficiency and efficacy.

7.7 Summary

In this chapter, we have presented a set of metamorphic relations on feature models and an automated test data generator based on them. Given a feature model and its set of products, our tool generates neighbouring models and their corresponding set of products. Generated products are then inspected to obtain the expected output of a number of analysis operations over the models. Non-trivial feature models representing millions of products can be efficiently generated applying this process iteratively. In order to evaluate our approach, we checked the effectiveness of our tool in detecting faults using mutation testing as well as real faults and tools. Two defects were detected in a recent release of FaMa and another two in SPLOT, an online feature model analyser actively used by the community. We also showed how our generator outperforms our manual suite for the analysis of feature models. Finally, we explained how our approach can be refined by using a set of initial test cases that guide the generation of test data improving the detection of faults. Our results show that the application of metamorphic testing in the domain of automated analysis of feature models is efficient and effective in detecting most faults in a few seconds without the need for a human oracle. To the best of our knowledge, this is the first automated approach for functional testing on the analyses of feature models.

The main results of this chapter were presented in the Third International Conference on Software Testing, Verification and Validation (ICST'10) [151]. An extension of that paper including the comparison and refinement with our manual suite was presented in the Information and Software Technology journal [153]. Finally, we reported on our experience using mutation testing with the FaMa framework in a paper accepted for publication in the Information and Software Technology special issue on Mutation Testing [152].

Chapter 8

Automated generation of hard feature models

If it ain't broke, you are not trying hard enough.

Popular saying

The rapid progress on the analysis of feature models is leading to an increasing interest to test and compare the performance of analysis solutions. One of the main challenges in this scenario is to find hard input models that show the behaviour of the tools in extreme situations (e.g. those producing longest execution times or highest memory consumption). Currently, these feature models are generated randomly ignoring the internal aspects of the tools under tests. As a result, these only provide a rough idea of the behaviour of the tools with average problems and are not sufficient to reveal their real strengths and weaknesses. In this chapter, we model the problem of finding computationally–hard feature models as an optimization problem and we solve it using a novel evolutionary algorithm. Given a tool and an analysis operation, our algorithm generates input models of a predefined size maximizing aspects as the execution time or the memory consumption of the tool when performing the operation over the model. This allows users and developers to know the behaviour of tools in pessimistic cases providing a better idea of their real power. Experiments using our evolutionary algorithm on a number of analysis operations and tools have successfully identified input models causing much longer executions times and higher memory consumption than random models of identical or even larger size.

The rest of the chapter is structured as follows: In Section §8.1, we motivate the problem addressed and introduce the solution proposed. In Section §8.2, we present a novel evolutionary algorithm to deal with optimization problems on feature models and show how it can be applied to the search of computationally–hard feature models. The empirical evaluation of our approach is presented in Section §8.3. Section §8.4 presents the threats to validity of our work. Finally, we summarize our conclusions in Section §8.5.

8.1 Introduction

As presented in our analysis of current solutions (see Chapter §5), recent publications reflect an increasing interest to evaluate and compare the performance of analysis techniques and tools on the analyses of feature models. One of the main challenges when assessing performance is to find hard problems that show the strengths and weaknesses of the tools under test in extreme situations (e.g. those producing longest execution times). Feature models from real domains are by the far the most appealing input problems. Unfortunately, although there are references to large–scale real feature models, only small examples from research publications or case studies are available. For instance, the largest feature model available in the SPLOT feature model repository [159] at the time of writing this paper has 287 features. This lack of hard realistic feature models, has led authors to evaluate their tool with large–scale randomly generated feature models of 5000, 10000 and up to 20000 features. More recently, some authors have also suggested looking for tough and realistic feature models into the open source community.

Regardless of the type of feature model used during experimentation, the characteristics of the tools under tests are not considered in the current state of the art. As a result, current performance evaluations only provide a rough idea of the behaviour of tools with average problems rather than looking for specific weak points related to the type of technique or algorithm under evaluation. Hence, developers and users would probably be more interested to know whether their tool can crash with a hard realistic model of small or medium size rather than knowing the execution times of huge random model out of their scope.

The main goal of software testing is to find inputs that reveal errors in the software under test. The exhaustive search for these inputs is acknowledged to be unfeasible due to the size and complexity of the programs, there are simply too many inputs combinations. As pointed by McMinn [105], random testing is not a feasible solution: *"random methods are unreliable and unlikely to exercise 'deeper' features of software that are not exercise by mere chance"*. In this context, metaheuristic search techniques have proved to be a promising solution for the automated generation of test data for both functional [105] and non–functional properties [3]. *Metaheuristic search techniques* are frameworks which use heuristics to find solutions to hard problems at an affordable computational cost. Typical metaheuristic techniques are evolutionary algorithms, hill climbing or simulated annealing [182]. For the generation of test data, these strategies translate the test criteria into an objective function (also called fitness function) that is used to evaluate and compare the candidate solutions respect to the overall search goal. Using this information, the search is guided toward promising areas of the search space. Wegener et al. [187, 188] were one of the first proposing using evolutionary algorithms to verify the time constraints of software back in 1996. In their work, the authors used genetic algorithms to find input combinations that violate the time constraints of real–time systems, that is, those inputs producing an output too early or too late. Their experimental results showed that evolutionary algorithms are much more effective than random search in finding input combinations maximizing or minimizing execution times. Since then, a number of authors have followed their steps using metaheuristics and especially evolutionary algorithms for the testing of non–functional properties such as execution time, quality of service, security, usability or safety [3, 105].

Inspired by the ideas of Wegener and later authors, in this chapter we propose using evolutionary algorithms for the automated generation of hard feature models. In particular, we model the problem of finding computationally–hard feature models as an optimization problem and we solve it using a novel evolutionary algorithm for feature models. Given a tool and analysis operation, our algorithm generates input models of a predefined size maximizing aspects as the executions times or the memory consumption of the tool when performing the operation. For the evaluation of our approach, we performed several experiments using different analysis operations, paradigms, tools and optimization criteria. In total, we performed over 50 million executions of analysis operations for the configuration and evaluation of our approach. The results showed how our evolutionary program successfully identified input models causing much longer executions times and higher memory consumption than random models of identical or even larger size. Furthermore, the data revealed that the hard feature models found have similar properties to the realistic models found in the literature.

Our work enhances and complements the current state of the art of performance testing of feature model analysis tools as follows:

- Our approach is the first one using a search–based strategy to exploit the internal weaknesses of the analysis tools and techniques under evaluation rather than trying to detect them by chance using random models.
- Our work allows developers to focus on the search of computationally–hard models of realistic size that could reveal deficiencies in their tools rather than using huge feature models out of their scope.
- Our approach provides users with helpful information about the behaviour of tools in pessimistic cases helping them to choose the tool that better adapts to their needs.
- Our approach is highly generic being applicable to any automated operation on feature models, not only analyses, in which the quality (i.e. fitness) of the models with respect to an optimization criteria can be measured quantitatively.
- Our experimental results show that the hardness of feature models depends on different factors in contrast to related works in which the complexity of the models is mainly associated to their size. Although this is generally true, our work demystifies the belief that large models have to be necessarily harder to process than small ones.

8.2 Automated generation of hard feature models

In this section, we present the core of our contribution. First, we introduce a novel evolutionary algorithm to deal with optimization problems on feature models. Then, we present a specific instantiation of the algorithm to search for computationally-hard feature models.

8.2.1 An evolutionary algorithm for feature models

In this section, we present a novel evolutionary algorithm for solving optimization problems on feature models. The algorithm takes several size constraints and a fitness function as input and returns a feature model of the given size maximizing the optimization criteria defined by the function. In Section §4.4.1, we described the general structure of an evolutionary algorithm and explained its basic steps. In the following, we describe how these basic steps are carried out in our algorithm.

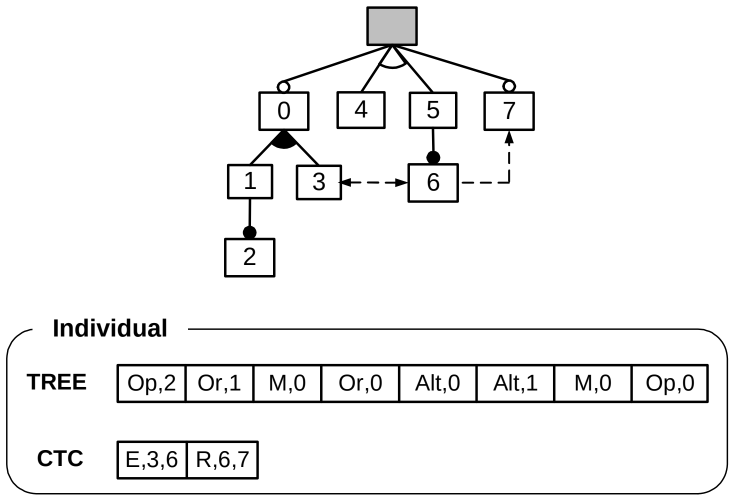


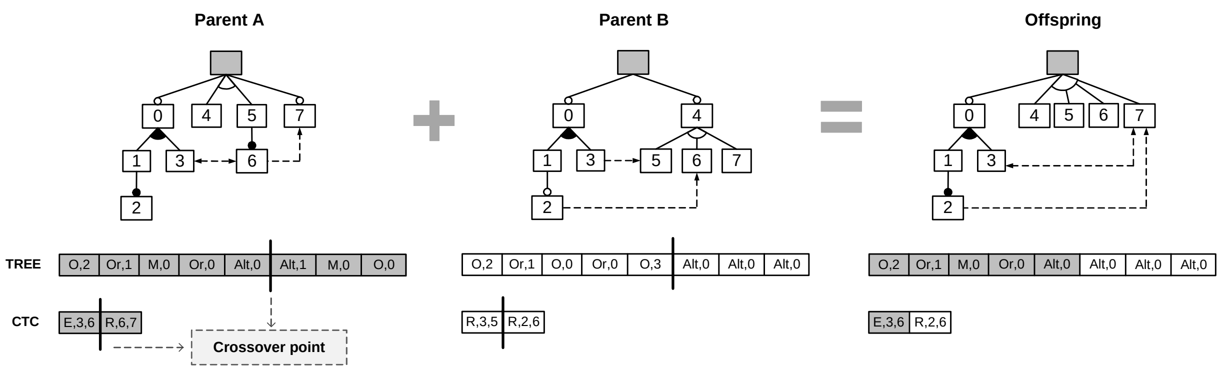
Figure 8.1: *Encoding of a feature model.*

Initial population. The initial population is generated randomly according to the size constraints received as input. The current version of our algorithm allows the user to specify the number of features, percentage of cross-tree constraints and maximum branching factor of the feature model to be generated.

Evaluation. Feature models are evaluated according to the fitness function received as input obtaining a numeric value that represents the quality of the candidate solution (i.e. its fitness).

Encoding. For the representation of feature models as individuals we propose using a custom encoding. Usual encodings for evolutionary algorithms were ruled out since these were either not adequate to represent tree structures (e.g. binary encoding [10]) or were not able to produce solutions of a fixed size (e.g. tree encoding [92]), a key requirement in our approach. Figure §8.1 depicts an example of our encoding. As illustrated, each model is represented by means of two arrays, one storing information about the tree and another one with information about Cross-Tree Constraints (CTC). The order of each feature in the array corresponds to the Depth–First Traversal (DFT) order of the tree. Hence, feature labelled with *'0'* in the tree is stored in the first position of the array, feature labelled with *'1'* is stored the second position and so on. Each feature in the tree array is defined as a two-tuple $< \mathtt{PR}, \mathtt{C} >$ where *PR* is the type of relationship with its parent feature (M: Mandatory, Op: Optional, Or: Or-relationship, Alt: Alternative) and *C* is the number of children of the given feature. As an example, first position in the tree array, $< \mathtt{Op}, 2 >$, indicates that feature labelled with *'0'* in the tree has an optional relationship with its parent feature and has two child features (those labelled with *'1'* and *'3'*). Analogously, each position in the CTC array stores information about one constraint in the form $< \mathtt{TC}, \mathtt{O}, \mathtt{D} >$ where *TC* is the type of constraint (R: Requires, E: Excludes) and *O* and *D* are the indexes of the origin and destination features in the tree array respectively.

Selection. This step determines how the individuals of one generation are selected to be combined and produce new offspring. Selection strategies are generic and can be applied regardless of how the individuals are represented. In our algorithm, we experimented with both rank-based roulette-wheel and binary tournament selection strategies obtaining positive results with both of them (see Section §4.4.1 for details about these techniques).



Parent A
Parent B
Offspring
TREE
CTC
O,2 | Or,1 | M,0 | Or,0 | Alt,0 | Alt,1 | M,0 | O,0
E,3,6 | R,6,7
Crossover point
O,2 | Or,1 | O,0 | Or,0 | O,3 | Alt,0 | Alt,0 | Alt,0
R,3,5 | R,2,6
O,2 | Or,1 | M,0 | Or,0 | Alt,0 | Alt,0 | Alt,0 | Alt,0
E,3,6 | R,2,6

Figure 8.2: *Example of one-point crossover in our algorithm.*

Crossover. These are the techniques used to combine chromosomes in some way and produce new individuals in an analogous way to biological reproduction. We tried two different crossover techniques in our algorithm with positive results, one–point and uniform crossover. Figure §8.2 depicts an example of the application of one–point crossover in our algorithm. The process starts by selecting two parent chromosomes (i.e. encoded feature models) to be combined. For each array in the chromosomes, the tree and CTC arrays, a random point is chosen (so–called crossover point). Finally, the offspring is created by copying the content of the arrays from the beginning to the crossover point from one parent and the rest from the other one. Notice that the characteristics of our encoding guarantee a fixed size for the individuals.

Mutation. In this step, random changes are applied to the chromosomes to prevent the algorithm from getting stuck prematurely at a locally optimal solution. Mutation operators must be specifically designed for the type of encoding used. In our algorithm, we defined four different types of custom mutation operators, namely:

- *Operator 1.* It changes randomly the type of a relationship in the tree array, e.g. from mandatory,$< \mathbf{M}, 3 >$, to optional,$< \mathbf{Op}, 3 >$.
- *Operator 2.* It changes randomly the number of children of a feature in the tree, e.g. from $< M, \mathbf{3} >$ to $< M, \mathbf{5} >$. The new number of children is in the range $[0, BF]$ where *BF* is the maximum branching factor indicated as input.
- *Operator 3.* It changes the type of a cross-tree constraint in the CTC array, e.g. from excludes $< \mathbf{E}, 3, 6 >$ to requires $< \mathbf{R}, 3, 6 >$.
- *Operator 4.* It changes randomly (with equal probability) the origin or destination feature of a constraint in the CTC array, e.g. from $< E, \mathbf{3}, 6 >$ to requires $< E, \mathbf{1}, 6 >$. Origin and destination features are ensured to be different.

These operators are applied randomly with the same probability.

Decoding. At this stage, the array-based chromosomes are translated back into feature models in order to be evaluated. In our algorithm, we identified three types of patterns making a chromosome infeasible or semantically redundant, namely: i) those encoding set relationships (or- and alternative) with a single child feature (e.g. Figure §8.3(a)), ii) those containing cross-tree

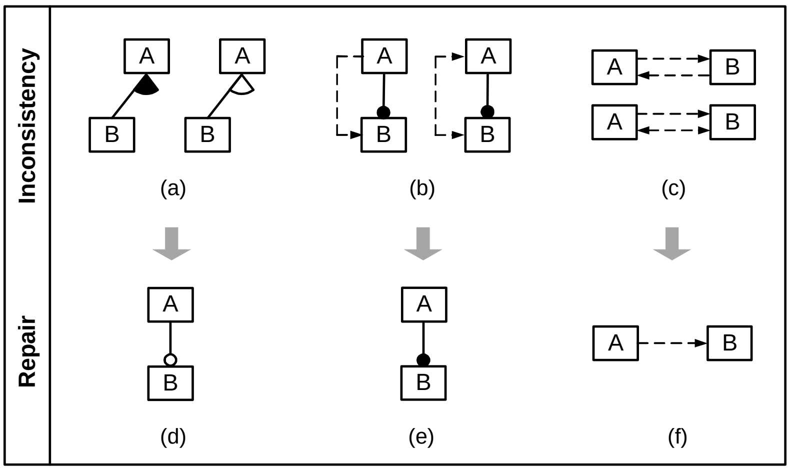


Figure 8.3: *Examples of infeasible individuals and repairs.*

constraints between features with parental relationship (e.g. Figure §8.3(b)), and iii) those containing features sharing contradictory or redundant cross-tree constraints (e.g. Figure §8.3(c)). The specific approach used to address infeasible individuals, replacement or repairing, mainly depend on the problem and it is ultimately up to the user.

Survival. Finally, the next population is created by including all the new offspring plus those individuals from the previous generation that were selected for crossover but did not generate descendants due to probability.

8.2.2 Instantiation of the algorithm

In this section, we propose to model the problem of finding computationally–hard feature models as an optimization problem and to solve it using an instantiation of our evolutionary algorithm. We chose evolutionary computation because it has proved to be a robust search technique suited for the complex search spaces and noisy objective functions used when dealing with non–functional properties [3]. Key benefit of our approach is that it takes into account the characteristics of the tools under test trying to exploit its vulnerabilities. Also, our approach is very generic being applicable to any automated operation on feature models, not only analyses, in which the quality (i.e. fitness) of the models can be measured quantitatively.

In order to find a suitable configuration of our algorithm, we performed numerous executions of a sample optimization problem evaluating different combination of values for the key parameters of the algorithm, presented in Table §8.1. The optimization problem was to find a feature model maximizing the execution time invested by the analysis tool when checking whether the model is void (i.e. whether it represents at least one product). We chose this analysis operation because it is currently the most quoted in the literature [21]. In particular, we looked for feature models of different size maximizing execution time in the CSP solver JaCoP integrated into the FaMa framework v1.0. We choose FaMa mainly because our familiarity with the tool. Next, we clarify the main aspects of the configuration of our algorithm:

- **Fitness function.** Our first attempt was to measure the execution time in milliseconds invested by FaMa to perform the operation. However, we found that this was very inaccurate since the result of the function was deeply affected by the system load, i.e. it was not deterministic. To solve this problem, we decided to measure the fitness of a feature model as the number of backtracks produced by the analysis tool during its analysis. A *backtrack* represents a partial candidate solution to a problem that is discarded because it cannot be extended to a full valid solution [173]. In contrast to the execution time, most CSP backtracking heuristics are deterministic. Together with execution time, the number of backtracks is commonly used to measure the complexity of constraint satisfaction problems [173]. Thus, we may assume that the higher the number of backtracks the longer the computation time.
- **Infeasible individuals.** We evaluated the effectiveness of both replacement and repairing techniques and we finally opted for the later. More specifically, we used the following repairing algorithm with infeasible individuals: i) isolated set relationships are converted into optional relationships (e.g. the model in Figure §8.3(a) is changed as in Figure §8.3(d)), ii) cross-tree constraints between features with parental relationships are removed (e.g. the model in Figure §8.3(b) is changed as in Figure §8.3(e)), and iii) two features cannot share more than one constraint (e.g. the model in Figure §8.3(c) is changed as in Figure §8.3(f)).
- **Stop criteria.** There is no means of deciding when an optimum input has been found and the evolutionary algorithm should be stopped [188]. Therefore, we decided to allow the algorithm to continue for a given number of executions of the fitness function taking the largest number of backtracks obtained as the optimum, i.e. solution to the problem.

Table §8.1 depicts the values evaluated for each parameter. These values were based on related works with evolutionary algorithms and our previous experience in this domain. Each combination of parameters was executed 10 times to avoid heterogeneous results and perform statistical analysis of the data. Underlined values were those providing better results and therefore those selected for the final configuration of our algorithm. In total, we performed over 40 million executions of the objective function to find a good setup for our algorithm. Statistical significance test of the effect of parameters on best objective function value obtained for each parameter setting were performed on results obtained, in order to ensure soundness of selected values. When test shown non-significant differences on performance, values with the best average performance were chosen.

Parameter	**Values evaluated and selected**
Selection strategy	Roulette-wheel, 2-Tournament
Crossover strategy	One-point, Uniform
Crossover probability	0.7, 0.8, 0.9
Mutation probability	0.0075, 0.005, 0.02
Size initial population	50, 100, 200
#Executions fitness function	2000, 5000
Infeasible individuals	Replacing, Repairing

Table 8.1: *Algorithm configuration.*

8.3 Evaluation

In order to evaluate our approach, we developed a prototype implementation of our algorithm in Java. This prototype is available as a part of the BeTTy framework described in Appendix §A.

In general, it is not possible to verify that the solution obtained by the algorithm represents a global optimum. Although there are static techniques that could be used for this (e.g. control flow graph analysis), these are not affordable in general for large complex software [188]. Thus, we decided to evaluate the efficacy of our approach by comparing it to random search since this is the most extended strategy for performance testing in the current state of the art. In particular, the evaluation of our evolutionary program was performed through a number of experiments. On each experiment, we compared the effectiveness of random generators and our evolutionary program on the search of feature models maximizing properties such as the execution time or memory consumption required for their analysis. Additionally, we performed some extra experiments studying the characteristics of the hard feature models generated and the behaviour of the program with different stop criteria.

All the experiments were performed on a desktop machine equipped with an Intel Xeon CPU 5140@2.33GHz running Windows Server 2003 and Java 1.6.0_13 on 1400 MB of dedicated memory.

8.3.1 Experiment #1: Maximizing execution time

In this experiment, we evaluated the ability of our evolutionary algorithm to search input feature models maximizing the analysis time of a solver. In particular, we measured the execution time required by a CSP solver to find out if the input model is consistent (i.e. whether it is void or not). Notice that this was the same problem used to tune the configuration of our algorithm. Again, we chose the consistency operation because it is currently the most used in the literature. Next, we present the setup and results of our experiment.

Experimental setup. This experiment was performed through a number of iterative steps. On each step, we generated 5,000 random feature models and checked their consistency saving the maximum fitness obtained. Then, we executed our evolutionary program and allowed it to run during the same number of executions of the fitness function (i.e. 5,000) and compared the results. This process was repeated with different size of the models to evaluate the scalability of our algorithm. In particular, we generated models with different combinations of features, {200, 400, 600, 800, 1,000} and percentage of constraints (with respect to the number of features), {10%, 20%, 30%, 40%}. Additionally, for each size of the model, we repeated the process 20 times to get averages and perform statistical analysis of the data. In total, we performed 4 million executions of the fitness function for this experiment. The fitness was set equal to the number of backtracks obtained by the analysis tool when checking the model consistency. For the analysis, we used the solver JaCoP integrated into FaMa v1.0 with the default heuristics *MostConstrainedDynamic* for the selection of variables and *IndomainMin* for the selection of values from the domains. To prevent the experiment from getting stuck, a maximum timeout of 30 minutes was used for the execution of the fitness function in both the random and evolutionary search. If this time was exceeded, a new iteration was started. After all the executions, we measured the execution time of the hardest feature models found for a full comparison, i.e.

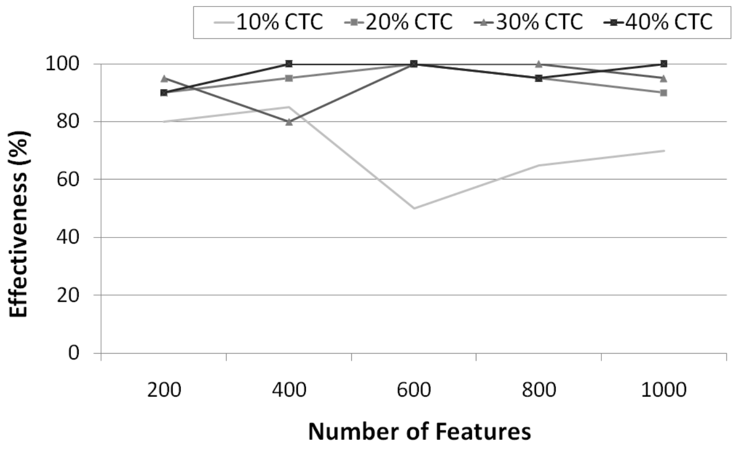


Figure 8.4: *Effectiveness of the evolutionary algorithm.*

those producing a larger number of backtracks. More specifically, we executed 10 times each optimal solution to get average execution times.

Analysis of results. Figure §8.4 depicts the effectiveness of our algorithm for each size range of the feature models generated. We define the *effectiveness* of our evolutionary program as the percentage of times (out of 20) in which the program found a better optimum than random models, i.e. a higher number of backtracks. As illustrated, the effectiveness of evolutionary programming was over 90% in most of the cases reaching 100% in six of them. Overall, our evolutionary program found harder feature models than those generated randomly in 88.75% of the executions. We may remark that our algorithm revealed the lowest effectiveness with those models containing 10% of cross-tree constraints. We found that this was due to the simplicity of the analysis in these models. The number of backtracks produced by these models was very low, zero in most of the cases, and thus our evolutionary program had problems finding promising individuals that could evolve towards optimal solutions.

Table §8.2 depicts the evaluation results for the range of feature models with 20% of cross-tree constraints. For each number of features and search technique, random and evolutionary, the table shows the average and maximum fitness obtained as well as the average and maximum execution times of the hardest feature models found. The effectiveness of the evolutionary program is also presented in the last column. As illustrated, the evolutionary program found feature models producing a number of backtracks larger by several orders of magnitude than those produced by random models. The fitness of the hardest models generated using our evolutionary approach was on average 1,950 times higher than that of random models (108,579.13 backtracks against 55.63) and 2,450 times higher in the maximum value (6.6 million backtracks against 2,751). As expected, these results were also reflected in the execution times. On average, the CSP solver invested 0.03 seconds to analyse the random models and 5.35 seconds to analyse those generated using our evolutionary generator. The superiority of evolutionary search was especially remarkable in the maximum times ranging from the 0.23 seconds of random models to the 251.45 seconds (4.1 minutes) invested by the CSP solver to analyse the hardest feature model generated by our evolutionary program. Overall, our evolutionary approach produced a harder feature model than random techniques in 94% of the executions in the range of 20% of constraints.

	Random Testing				Evolutionary Algorithm				
#Feat.	**Avg Fitness**	**Max Fitness**	**Avg Time (s)**	**Max Time (s)**	**Avg Fitness**	**Max Fitness**	**Avg Time (s)**	**Max Time (s)**	**Effectiveness**
200	6.45	15	0.01	0.01	310.25	2,122	0.02	0.06	90
400	13.20	37	0.02	0.02	8,028.85	153,599	0.28	4.80	95
600	29.50	223	0.03	0.06	8,765.65	118,848	0.67	7.33	100
800	53.95	304	0.05	0.06	346,217.95	6,678,168	13.19	251.45	95
1,000	175.05	2,751	0.07	0.23	179,572.95	3,167,253	12.58	208.91	90
Total	**55.63**	**2,751**	**0.03**	**0.23**	**108,579.13**	**6,678,168**	**5.35**	**251.45**	**94**

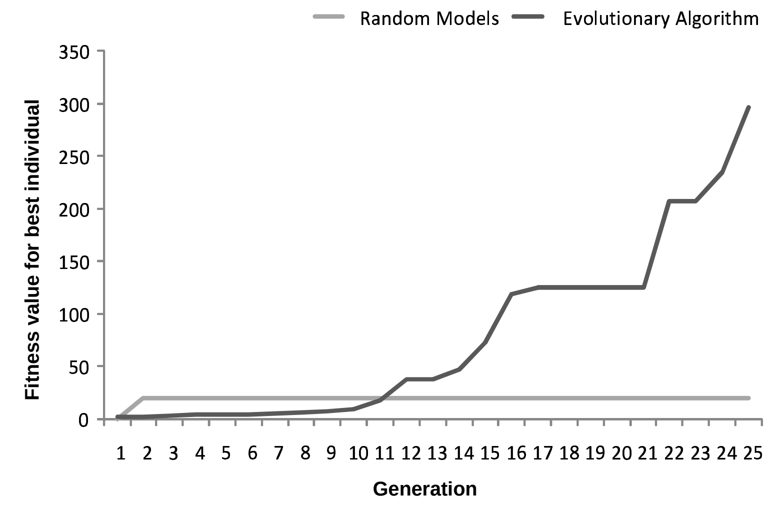
Table 8.2: *Evaluation results on the generation of feature models maximizing execution time.*

A global summary of the results is presented in Table §8.3. The table depicts the maximum execution times invested by the CSP solver to analyse the hardest models found using random and evolutionary search. The data show that our approach was more effective than random models in all size ranges. The hardest random model required 0.23 seconds to be processed. In contrast, our evolutionary approach found five models requiring between 1 and 4.2 minutes to be analysed. Interestingly, our algorithm was able to find harder but significantly smaller feature models (between 400 and 800 features) than the hardest random model found (1,000 features). This emphasizes the ability of our approach to generate motivating input models of realistic size that reveal the vulnerabilities of tools and heuristics instead of just running them using large random models.

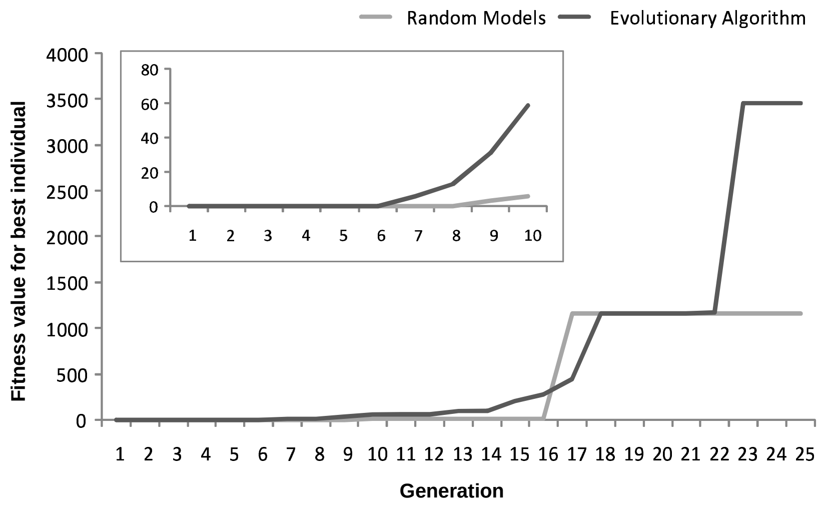
	10% CTC		20% CTC		30% CTC		40% CTC	
#Feat.	**Rand. Time (s)**	**EA Time (s)**	**Rand. Time (s)**	**EA Time (s)**	**Rand. Time (s)**	**EA Time (s)**	**Rand. Time (s)**	**EA Time (s)**
200	0.01	0.05	0.01	0.06	0.02	0.14	0.01	0.02
400	0.07	0.22	0.02	4.80	0.02	1.02	0.02	0.10
600	0.05	1.78	0.06	7.33	0.03	4.79	0.03	7.25
800	0.05	62.30	0.06	251.45	0.05	250.95	0.05	0.35
1,000	0.10	3.43	0.23	208.91	0.07	84.99	0.06	0.49

Table 8.3: *Maximum execution times produced by random and evolutionary search.*

Figure §8.5 compares random and evolutionary techniques for the search of a feature model maximizing the number of backtracks in two sample executions. We may remark that it was very hard to find two motivating examples that could be represented graphically. This occurred because the results obtained by our evolutionary program were so much higher than those of random models that it was unfeasible to represent them using a similar scale. Horizontally, the graphs show the number of generations where each generation represent 200 executions of the fitness function. Figure §8.5(a) shows that random models reaches its maximum number of backtracks after only about 400 executions. That is, the generation of 4,600 other random models do not produce any higher number of backtracks and therefore are useless. In contrast to this, our evolutionary approach shows a continuous improvement up to the 25th generation. After 11 generations (about 2,200 executions), the fitness found by evolutionary search are above of those of random models. Figure §8.5(b) depicts another example in which random models are lucky to find a high number of backtracks in the 17th generation. Evolutionary optimization, however, once again manages to improve the execution times continuously overcoming the best random fitness after 22 generations. In generation number 23, even a significant



a) Feature models with 200 features and 40% of cross-tree constraints



b) Feature models with 400 features and 10% of cross-tree constraints

Figure 8.5: *Comparison of random models and our evolutionary algorithm for the search of the highest number of backtracks.*

leap of more than 2,000 backtracks can be observed. In both examples, the curve trace suggests that the evolutionary algorithm would find even better solutions if the number of generations were increased. This was confirmed in a later experiment in which the program was allowed to run for up to 125 generations (25,000 executions of the fitness function) finding feature models producing more than 13 million backtracks (see Section §8.3.3 for details).

8.3.2 Experiment #2: Maximizing memory consumption

In this experiment, we evaluated the ability of our evolutionary program to generate input feature models maximizing the memory consumption of a solver. In particular, we measured the memory consumed by a BDD solver when finding out the number of products represented by the model. We chose this analysis operation because it is one of the hardest operations in terms of complexity and it is currently the second operation most quoted in the literature [21]. We decided to use a BDD-based reasoner for this experiment since it has proved to be the most efficient option to perform this operation [21].

Experimental setup. This experiment consisted of a number of iterative steps. At each step, we generated 5,000 random models and compiled each of them into a BDD for counting the number of solutions measuring its size. We then executed our evolutionary program and allowed it to run for 5,000 executions of the fitness function looking for feature models maximizing the size of the BDD and compared the results. Again, this process was repeated with different combination of features, {50, 100, 150, 200, 250} and percentage of constraints, {10%, 20%, 30%} to evaluate the scalability of our approach. For each size of the model, we repeated the process 20 times to get statistics from the data. In total, we performed about 3 million executions of the fitness function for this experiment. We may remark that we generated smaller feature models than those presented in previous experiment in order to reduce BDD building time and make the experiment affordable. Measuring memory usage in Java is difficult and computationally expensive since memory profilers usually add a significant overload to the system. To simplify the fitness function, we decided to measure the fitness of a model as the number of nodes of the BDD representing it. This is a natural option used in the research community to compare the space complexity of BDD tools and heuristics [110]. For the analysis, we used the solver JavaBDD integrated into the feature model analysis tool SPLOT. We chose SPLOT because it integrates highly efficient ordering heuristics specifically designed for the analysis of feature models using BDDs. In particular, we used the heuristic *'Pre-CL-MinSpan'* presented by Mendonca et al. in [110]. As in our previous experiment, we set a maximum timeout of 30 minutes for the fitness function to prevent the experiment from getting stuck when finding too good solutions. We found, however, that this timeout was constantly exceeded by the feature models found by our algorithm in the range of 250 features and 30% of constraints. To simplify the evaluation in this size range, we made our algorithm stop as soon as it found a better fitness than those produced by random models.

Analysis of results. Table §8.4 depicts the number of BDD nodes of the hardest feature models found using random techniques and our evolutionary program. For each size range, the table also shows the computation time (BDD building time + execution time) invested by SPLOT to analyse the model. As illustrated, our evolutionary program found better results than random techniques in all size ranges. On average, the BDD size found by our evolutionary approach was between 2 and 12.5 times higher than those obtained with random models. The largest BDD generated from random models had 25.3 million nodes while the largest BDD obtained using our evolutionary program had 27.9 million nodes. The results suggest, however, that the maximum found by evolutionary search would be much higher if we would not have limited the improvement factor in the range of 250 features (30% constraints) to make the experiment affordable. As expected, the superiority of our evolutionary program was also observed in the computation times required by each model to be compiled and analysed. This suggests that our approach can also deal with optimization criteria involving compilation time. Overall, our evolutionary program found feature models producing higher memory consumption than random models in 99.3% of the executions.

	10% CTC				20% CTC				30% CTC			
	Random		Evolutionary		Random		Evolutionary		Random		Evolutionary	
#Feat.	**BDD size**	**Time (s)**	**BDD Size**	**Time (s)**	**BDD size**	**Time (s)**	**BDD Size**	**Time (s)**	**BDD size**	**Time (s)**	**BDD Size**	**Time (s)**
50	781	0	1,963	0	2,074	0	8,252	0.01	2,455	0.01	10,992	0.01
100	7,629	0.01	20,077	0.02	33,522	0.03	161,157	0.20	95,587	0.08	419,835	0.73
150	65,627	0.10	188,985	0.31	374,675	0.91	3,060,590	12.80	673,410	1.28	11,221,303	24.22
200	203,041	0.09	924,832	0.86	2,735,005	4.34	19,698,780	75.05	3,394,435	58.22	23,398,161	380.52
250	1,720,983	3.69	7,170,121	25.94	25,392,597	82.28	27,970,630	253.32	20,579,015	343.72	22,310,416	431.62

Table 8.4: *BDD size and computation time of the hardest feature models found using random techniques and our evolutionary program.*

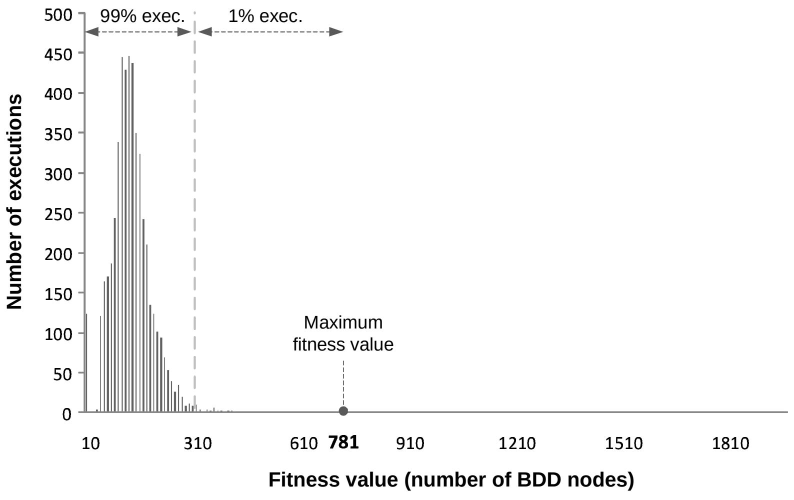
Figure §8.6 shows the frequency with which each fitness value was found during the search of a feature model producing the largest BDD. The data presented corresponds to the hardest feature models generated in the range of 50 features and 10% of cross-tree constraints. We chose this size range because it produced the smallest BDD sizes and facilitated the comparison of the results of both techniques using the same scale. For random models (Figure §8.6(a)), a narrow Gaussian-like curve is obtained with more than 99% of the executions producing fitness values under 300 BDD nodes. During evolutionary execution (Figure §8.6(b)), however, a wider curve is obtained with 39% of the execution producing values over 300 nodes. Both histograms clearly show how evolutionary programming performed a more exhaustive search in a larger portion of the solution space than that explored by random models. This trend was also observed in the rest of size ranges.

During this experiment, we found that the fitness function was not deterministic, that is, different executions with the same input feature model produced different number of BDD nodes. We found, however, that the variations in the number of nodes were small and did not affect the effectiveness of our evolutionary program.

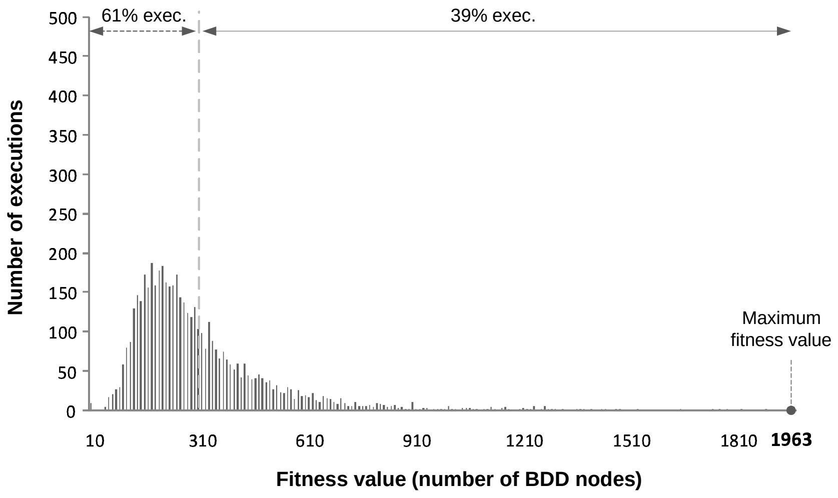
8.3.3 Additional results and discussion

As a part of our evaluation, we performed some extra experiments to be reported in an external technical report. Among other results, we studied the ability of our algorithm to generate input models maximizing execution time in a SAT solver. The setup and results of this experiment were similar to those presented in Sections §8.3.1 and §8.3.2. In the experiment, our evolutionary approach succeeded in finding harder feature models than those generated randomly in 91.2% of the executions. We may remark, however, that the differences in the execution times obtained using random and evolutionary techniques were not significant. This was expected since it has been proved that the analysis of feature models with simple cross-tree constraints (i.e. those involving three features or less) using SAT solvers is highly efficient [109].

During our experimental work we noticed that the number of executions of the fitness function (i.e. stop criteria) had a great impact in the results of our evolutionary program. In order to evaluate this impact, we partially repeated Experiments #1 and #2 varying gradually the number of executions of the fitness function from 2,000 to 25,000. The results revealed that the effectiveness of our algorithm was over 90% when the number of executions was 5,000 or higher. More importantly, we found that the results provided by evolutionary search were better and better as the number of executions was increased without reaching a clear peak meanwhile the



a) Distribution of fitness values for random models



b) Distribution of fitness values for our evolutionary approach

Figure 8.6: *Distribution of fitness values for random and evolutionary search.*

results of random search showed little or no improvement at all. In the execution with the CSP solver, for instance, our evolutionary program produced a new maximum fitness of more than 13.2 million backtracks (obtained in 6.7 minutes) meanwhile random search found a maximum value of only 4,616 backtracks (obtained in 0.2 seconds). Similarly, the maximum fitness produced in our experiment with BDD in the range of 25,000 executions was five times higher than the maximum obtained when using 5,000 executions as stop criteria.

As a part our evaluation, we also studied the characteristics of the hardest feature models generated using our evolutionary approach in the experiments with CSP, SAT and BDD solvers, presented in Table §8.5. The data reveals that the models generated have a fair proportion of all different relationships and constraints. This is interesting since the algorithm is free to remove any type of relationship or constraints from the model if this helps to make it harder, but this did not happen in our experiments. We recall that the only constraints imposed by our algorithm are those regarding the number of features, number of constraints and maximum branching factor. Another piece of evidence is that differences between the minimum and maximum percentages of each modelling element are considerably small especially in the hardest models for the CSP solver. Furthermore, the average percentages found are very similar to those of feature models found in the literature. In [155], She et al. studied the characteristic of 32 published feature models and reported that they contain, on average, 25% of mandatory features (between 17.5% and 27.7% in our models), 44% of set subfeatures (between 37.8% and 41.8% in our models), 16% of set relationships (between 13.4% and 14.4% in our models), 6% of or-relationships (between 6.9% and 8.1% in our models) and 9% of alternative relationships (between 6.3% and 7% in our study). As a result, we conclude that the models generated by our algorithm are by no means unrealistic. On the contrary, in the context of our study, they are a fair reflection of the realistic models found in the literature. This suggests that the long execution times and high memory consumption detected by our algorithm could be therefore reproduced when using real models with the consequent negative effect in the user.

	CSP Solver			**SAT Solver**			**BDD Solver**		
Modelling element	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
% relative to no. of features									
Mandatory	26.8	27.7	29.2	20.5	25.0	27.5	8.0	17.5	23.2
Optional	32.0	33.8	34.9	34.0	35.8	39.0	33.6	37.8	40.0
Set subfeatures	36.0	37.8	39.5	36.3	38.5	40.0	37.5	41.8	48.0
Set relationships	13.5	14.2	14.5	9.5	13.4	14.6	13.5	14.4	16.0
- Or	6.7	7.1	7.5	5.0	6.9	7.5	7.2	8.1	10.0
- Alternative	6.5	7.0	7.4	4.5	6.5	7.2	6.0	6.3	7.6
% relative to no. of constraints									
Requires	44.7	49.7	61.1	45.0	49.4	54.7	25.0	45.5	52.1
Excludes	38.9	50.3	55.3	45.3	50.6	55.0	47.8	54.4	75.0

Table 8.5: *Statistics of the hardest feature models found in our experiments.*

In another experiment, we checked whether the hard feature models generated by our evolutionary approach were also hard for other heuristics. In particular, we repeated the analysis of the hardest feature models found in Experiment #1 using the other seven heuristics available in the CSP solver JaCoP. The results revealed that the hardest feature models found in our experiment, using the heuristic *MostConstrainedDynamic*, were trivially solved by some of the others heuristics. This finding supports our working hypothesis: feature models that are hard to analyse by one tool or technique could be trivially processed by others and vice-versa. Hence, we conclude that using standard set of problems, random or not, is therefore not sufficient for a full evaluation of the performance of different tools. Instead, as in our approach, the characteristics of the techniques and tools under evaluation must be carefully examined to identify their strengths and weaknesses providing helpful information for both users and developers.

Finally, we may remark that the effectiveness of our approach ranged from 88.7% to 99.2% in all the experiments. As expected from an evolutionary algorithm, we found that these variations in the effectiveness were caused by the characteristics of the search spaces of each problem.

In particular, the algorithm behaves better when the search space is heterogeneous and there are many different fitness values, i.e. it is easy to compare the quality of the individuals. However, results get worse in homogeneous search spaces in which most fitness values are equal. This was observed in Experiment #1 (range 10% of constraints) in which the number of backtracks was almost always zero hindering the search of promising individuals that could lead to optimal solutions. A common strategy to alleviate this problem is to use a larger population increasing the chances of the algorithm to find promising individuals during initialization.

For a more rigorous and exhaustive validation of our approach, we performed a statistical analysis on the experimental results. The data obtained from this analysis clearly confirmed the superiority of our algorithm when compared to random mechanisms. The description and results of our statistical study are presented in Appendix §D.

8.4 Threats to validity

We briefly discuss the threats to validity of our work:

- **Experimental procedure**. In order to ensure validity of the experimental approach, experiments were performed in a randomized order on the same computer and were replicated 20 times for each experimental configuration. Additionally, the results were formally validated by means of statistical tests that clearly showed the superiority of our algorithm when compared to random search. More specifically, as detailed in Appendix §D, Mann-Withney U tests were performed on the results obtained with random and evolutionary search.

- **Limitations of the approach**. Experiments showed no significant improvements when using our algorithm with problems of low complexity, i.e. feature models with 10% of constraints in Experiment #1. As stated in section §8.3.1, this limitation is due to the extremely flat shape of fitness landscape found in simple problems in which most fitness values are equal or close to zero. Another limitation of the experimental approach is that experiments for extremely hard feature models become unfeasible. We may remark, however, that this limitation is intrinsic to the problem of looking hard feature models and thus it also affects to random search. In fact, we emphasize that in the worst case our algorithm behave randomly equalling the strategies for the generation of hard feature models used in the current state of the art.

- **Generalizability of the conclusions**. In our experiments, we used two different analysis operations which could seem not sufficient to generalize the conclusions of our study. We remark, however, that these operations are currently the most quoted in the literature, have significantly different complexity and, more importantly, are the base for the implementation of many other analysis operations on feature models [21]. Thus, feature models that are hard to analyze for these operations would certainly be hard to analyse by those operations that use them as an auxiliary function making our results extensible to other analyses. Similarly, we just used two different analysis tools for the experiments, FaMa and SPLOT. We remark, however, that these tools are developed and maintained by independent laboratories providing the sufficient degree of heterogeneity for our study. Finally, the results obtained reveal that the shape and properties of the hard feature models generated are similar to those found in the literature and therefore there is no threat to validity due to the lack of realism of the generated models. We may mention that we imposed a constraint to our algorithm to make it simpler: features cannot have more than one alternative and one or relationship with their children. We remark, however, that this does not affect the feasibility of the results since the models generated are still realistic. Besides, we also imposed this constraint to the random feature models generated in order to make the results of our random and evolutionary programs comparable.

8.5 Summary

In this chapter, we have presented a novel evolutionary algorithm to solve optimization problems on feature models and showed how it can be used for the automated generation of computationally–hard feature models. Experiments using our evolutionary approach on different analysis operations and independent tools successfully identified input models producing much longer executions times and higher memory consumption than random models of identical or even larger size. In total, more than 50 million executions of analysis operations were performed to configure and evaluate our approach. When compared to previous works, our approach is the first one using a search–based strategy to exploit the internal weaknesses of the tools and techniques under test rather than simply using large–scale random models. This allows developers to focus on the search of tough models of realistic size that could reveal deficiencies in their tools rather than using huge feature models out of their scope. Similarly, users are provided with more information about the expected behaviour of the tools in pessimistic cases helping them to choose the tool or technique that better meets their needs. In view of the positive results obtained, we expect this work to be the first of many others research contributions exploiting the benefits of evolutionary computation in the field of feature modelling.

Part IV

Final Remarks

Chapter 9

Conclusions and future work

There will come a time when you believe everything is finished.
That will be the beginning.

Louis Dearborn, 1908–1988
American writer

9.1 Conclusions

The automated analysis of feature models is an active research topic. The extended use of feature models together with the many applications derived from their analysis has allowed this discipline to gain importance among researchers in software product lines. As a result, a number of analysis operations and approaches providing automated support for them are rapidly proliferating. In this context, the rapid progress of the discipline is leading to a increasing concern about the quality of analysis tools in terms of absence of faults and performance. However, current testing mechanisms in this domain are immature and guided by intuition rather than by well-established testing methods. This weakens the value and scope of research proposals and hinder the development of feature model analysis tools affecting their quality and reliability.

In this dissertation, we have presented a set of algorithms, techniques and tools to support functional and performance testing on the analysis of feature models. These contributions are the result of the application of popular techniques from the software testing community to the analysis of feature models. Our main results are a test suite, FaMa TeS, and a framework, BeTTy, enabling the efficient detection of faults in feature model analysis tools and the exhaustive study of their performance. To show the feasibility of our approach, we have presented extensive experimental results showing the efficacy and efficiency of our contributions. Among other results, we detected two faults in FaMa and another two in SPLOT, two popular analysis tools widely used in the community of automated analysis of feature models. These results support the success of our dissertation in providing a solid background for the development of automated support for the analysis of feature models contributing to the progress of the discipline and leading it to a new level of maturity.

9.2 Discussion, limitations and extensions

We next discuss some of the decisions that we have made in this dissertation highlighting its main limitations and possible extensions.

- **Feature modelling notations supported**. The testing techniques and tools presented in this dissertation were specifically designed to validate those analysis performed on basic feature models. (see Chapter §2). We chose this notation for its simplicity and extended use in the software product line community. In fact, basic feature models are by far the most used notation according to our literature review on the analysis of feature models [21]. Nevertheless, our review also showed that the analysis of cardinality–based and extended feature models is rapidly gaining popularity. Thus, the analysis support for these types of feature models could certainly benefit from the testing approaches presented in this dissertation.

  Extension: Extend the FaMa Test Suite and the test data generators integrated into BeTTy to support the testing of analysis tools dealing with cardinality–based and extended feature models.

- **Analysis operations supported**. The contributions presented in this dissertation support the testing of 16 out of the 30 analysis operations on feature models. These operations were mainly chosen for its extended use in the community of automated analysis [21]. However, there are still several operations with no specific testing support. This could probably hinder the implementation of these operations in feature model analysis tools.

  Extension: Extend the FaMa Test Suite and the BeTTy framework to support the testing of more analysis operations.

- **Test data generation constraints**. In testing, it is often useful to generate test data with specific properties. For instance, our metamorphic test data generator supports the generation of feature models with a specific number of features, percentage of each type of relationship, maximum branching factor, etc. Our evolutionary algorithm, however, deals with only a few configuration options for simplicity. Improving the flexibility of our algorithm would be desirable in order to deal with stricter structural constraints, e.g. enabling the generation of hard models with a given percentage of mandatory features.

  Extension: Refine our evolutionary algorithm to allow the generation of feature models with specific properties.

9.3 Other future work

In addition to the aforementioned extensions to our work, we identify a number of motivating topics to be explored in our future research, namely:

- **Metamorphic testing**. In this dissertation, we have presented a novel application of metamorphic testing enabling the automated generation of test data. However, we envision that the same procedure could be applied in a number of domains in which an input artefact is analysed to extract information from it. Consider, as an example, the CNF files used as input in SAT solvers.

Future work: Generalize our metamorphic testing approach and provide generic guidelines for the definition of metamorphic relations in similar domains.

- **Evolutionary testing**. We have presented a novel evolutionary algorithm for feature models and we have used it for the generation of computationally–hard feature models. However, further applications of our algorithm are still to be explored. Some promising applications are those dealing with the optimization of other non–functional properties in analysis operations. Additionally, it would also possible to use our approach to verify the time constraints of real time systems dealing with variability like those of mobile phones or context–aware pervasive systems. Also, we should study how our algorithm behaves when dealing with minimization problems. Last, but not least, we plan to study the hard feature model generated and try to understand what make them hard to analyse. From the information obtained, more refined applications and heuristics could be developed leading to a more efficient tool support for the analysis of feature models.

  Future work:

  ◇ Study new applications of our evolutionary algorithm dealing with other optimization criteria.

  ◇ Use our algorithm to verify the time constraints of real time system dealing with variability.

  ◇ Evaluate the behaviour of our algorithm when dealing with minimization problems.

  ◇ Study the characteristics of the hard feature model generated to understand what make them hard to analyse. Then, using the information obtained, design heuristics to determine which analysis solution is more efficient for a given input model.

- **Tool support**. The contributions presented in this dissertation have been integrated into a framework, BeTTy, supporting benchmarking and testing on the analysis of feature models. Extending the framework with new features will certainly be part of our future work. Some motivating features to be added are described below.

  Future work:

  ◇ Extend BeTTy to support the testing of analysis tools dealing with other variability models, e.g. Orthogonal Variability Models (OVMs) [129].

  ◇ Extend BeTTy with test data generators for cardinality–based and extended feature models.

  ◇ Implement an user Web interface for BeTTy.

- **Testing variability-intensive systems**. Current trends in software development such as Software Product Lines (SPL), Service-Oriented Applications (SOA) or Dynamically Adaptive Systems (DAS) respond to an increasing demand for highly configurable (and reconfigurable) software, i.e. so–called variability–intensive systems. These paradigms provide powerful mechanisms to manage variability but also pose important problems in terms of testing. One of the main challenges is that variability exponentially increases the number of tests required. As a consequence, new techniques and adequacy criteria should be defined. In this scenario, it would be interesting to know how the contributions presented in this dissertation could help in this endeavour.

  Future work: Study how the ideas presented in this dissertation could be applied or reused to address the testing of variability–intensive systems.

Part V

Appendices

Appendix A
BeTTy Framework

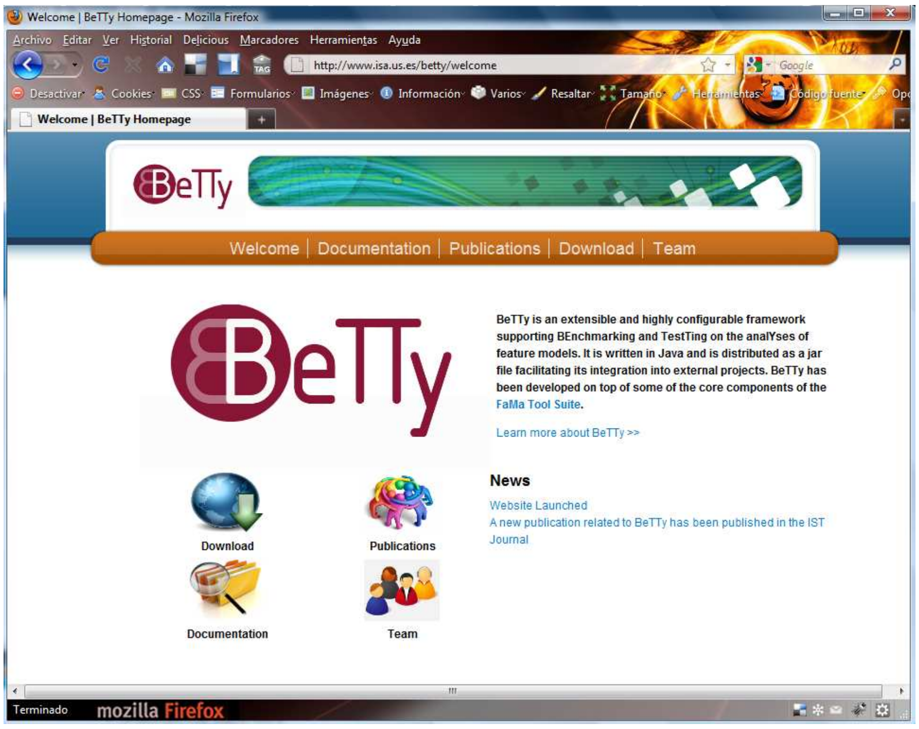
In this dissertation, we have motivated the need for automated support for both functional and performance testing of feature model analysis tools. In response to this need, we have presented several prototypes tools for the automated generation of test data. In order to make our contributions accessible to the feature modelling community and encourage other researchers and practitioners to use, evaluate and extend our work, we have integrated all these prototypes in a framework called BeTTy. This appendix describes the BeTTy framework. In Section §A.1, we provide a general overview of the tool. The architecture of BeTTy is explained in Section §A.2. Some fragments of source code illustrating the usage of the framework are listed in Section §A.3. In Section §A.4, we present the related tools found in the literature and compare them to BeTTy. Finally, a summary of the appendix is presented in Section §A.5.

A.1 General overview

BeTTy is an extensible and highly configurable tool supporting BEnchmarking and TestTing on the analYses of feature models. It is written in Java and is distributed as a jar file facilitating its integration into external projects. BeTTy has been developed on top of some of the core components of FaMa which make it part of the FaMa ecosystem. Current version of BeTTy provides the following features:

- **Random generation of basic feature models.** The generation can be highly configured through a number of input parameters such as number of features, percentage of cross–tree constraints, percentage of each type of relationship or maximum branching factor of the models to be generated. Users can optionally use a *"seed"* number to make the generation reproducible in later experiments.
- **Metamorphic generation of feature models and their products.** The framework allows users to generate not only random feature models but also the set of products that these represents. This is done transparently to the user by using the metamorphic relations presented in Section §7.2.1. The resultant model and set of products can be then used for functional testing as described in Chapter §7.
- **Evolutionary generation of feature models.** BeTTy integrates the evolutionary algorithm for feature model described in Chapter §8. This allows users to generate feature model maximizing or minimizing certain properties of the model or their analyses. For instance, we could search for a feature model with a given number of features minimizing the height of the tree or maximizing the execution time required for its analysis. This is done by simply defining an objective function and using it as an input of the framework. This function determines the quality of a feature model with respect to a given optimization criteria.
- **Feature model's characteristics extractor.** Given a feature model, BeTTy support the extraction of a good number of structural data from the model. These data include basic information such as the number of features or percentage of cross-tree constraints but also more complex data such as the cross–tree constraints ratio [21]. This feature allows users to examine the generated models and rule out those that do not fulfil certain properties.
- **FaMa feature model metamodel.** BeTTy integrates the feature model metamodel located at the core of the FaMa Ecosystem. This is a powerful implementation of a feature model metamodel highly tested and used in the feature modelling community. This component enables an intuitive representation of feature models and their manipulation.
- **Feature model readers and writers.** The framework supports loading and saving feature models and their products from/to files in different formats. Among others, these formats include XML, Simple XML feature model format (SXFM) [159] and dot format (for the visual representation of the models in the graph visualization tool Graphviz [69]).
- **Benchmarking.** BeTTy includes a set of classes to facilitate performance evaluation and comparison of feature model analysis tools. In particular, these classes provide support for the classical tasks of generating experiments, executing them and saving the results in format that facilitate their later processing in spreadsheets (e.g. Comma Separated Values, CSV). The framework also includes an instantiation of these classes for the FaMa framework making performance evaluations with this tool straightforward.

The BeTTy framework is freely distributed under LGPL v3 license and can be downloaded from the BeTTy Web site (see Figure §A.1).


Welcome | BeTTy Homepage - Mozilla Firefox
Archivo Editar Ver Historial Delicious Marcadores Herramientas Ayuda
http://www.isa.us.es/betty/welcome
Welcome | BeTTy Homepage
BeTTy
Welcome | Documentation | Publications | Download | Team
Download
Publications
Documentation
Team
Terminado
mozilla Firefox

BeTTy is an extensible and highly configurable framework supporting BEnchmarking and TestTing on the analYses of feature models. It is written in Java and is distributed as a jar file facilitating its integration into external projects. BeTTy has been developed on top of some of the core components of the FaMa Tool Suite.

Learn more about BeTTy >>

News

Website Launched
A new publication related to BeTTy has been published in the IST Journal

Figure A.1: *BeTTy Web Site (http://www.isa.us.es/betty).*

A.2 Architecture

Figure §A.2 presents a high level representation of the architecture of BeTTy . As illustrated, the framework is composed of two main blocks, the core and the extensions. The BeTTy core contains the set of interfaces and classes used to extend the framework and build applications on top of it. This mainly consists of the following components:

- *FaMa feature model meta model*. This is the set of classes used to represent feature models and manipulate them.
- *FM generation*. This component contains support methods and classes to facilitate the development of feature model generators.
- *Reader and writers*. This component consists of a set of ready–to–use feature model readers and writers for different formats.
- *Benchmarking*. This components contains the interfaces and basic classes to facilitate the generation, execution and saving of results during performance evaluation of analysis tools.

The BeTTy extensions comprise the set of testing and performance tools developed on top of BeTTy. These mainly consist of:

- *Random feature model generator*. This is a highly configurable random generator for feature models. The algorithm used for the generation is inspired in the one described by Thüm et al. [164].
- *Metamorphic feature model generator*. It supports the generation of feature models and their products using the metamorphic testing approach described in Chapter §7.
- *Evolutionary feature model generator*. This component allows the guided generation of feature models fulfilling a given optimization criteria. For its implementation, we used the evolutionary algorithm presented in Chapter §8.
- *FaMa Benchmark*. This consists of a set of classes to facilitate the performance evaluation of the feature model analysis tools integrated into the FaMa ecosystem.

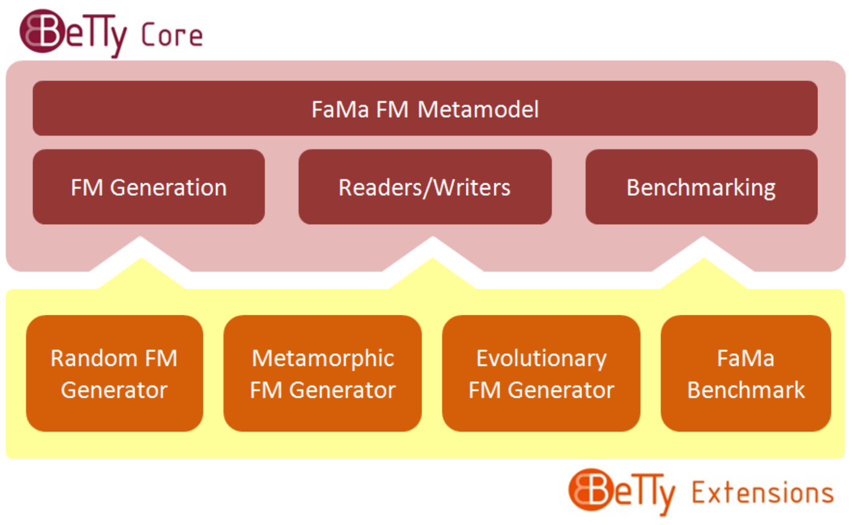


Figure A.2: *BeTTy framework architecture.*

A.3 Code examples

Listing §A.1 illustrates how to use BeTTy to generate a random feature model and save it in FaMa XML format (other formats are also available). In the example, the number of features and percentage of cross–tree constraints are given as input.

Listing A.1: Random feature model generation

```
public static void main(String[] args) throws Exception, BettyException {

  // STEP 1: Specify the user's preferences for the generation (so-called characteristics)
  GeneratorCharacteristics characteristics = new GeneratorCharacteristics();
  characteristics.setNumberOfFeatures(100);
```

```
 characteristics.setPercentageCTC(30);   .

 // STEP 2: Generate the model with the specific characteristics (FaMa FM metamodel is used)
 AbstractFMGenerator generator = new FMGenerator();
 FAMAFeatureModel fm = (FAMAFeatureModel) generator.generateFM(characteristics);

 // STEP 3: Save the model
 FMWriter writer = new FMWriter();
 writer.saveFM(fm, "./model.xml"); // Other valid formats: .splx, .afm, .dot
}
```

Listing §A.2 depicts a more detailed example in which both a random feature model and the set of products that it represents are generated. First, the generator object is created and a set input parameters are provided specifying the user's preferences for the generation. A reasonable limit for the number of products must be provided to avoid memory overflows. Once the generation has been completed, the code shows how to extract and print detailed statistics about the structural data of the model generated. Finally, both the model and its set of products are saved in XML format.

Listing A.2: Random generation of a feature model and its set of products

```
public static void main(String[] args) throws Exception, BettyException {

 // STEP 1: Specify the user's preferences for the generation (characteristics)
 GeneratorCharacteristics characteristics = new GeneratorCharacteristics();
 characteristics.setNumberOfFeatures(40);
 characteristics.setPercentageCTC(30);
 characteristics.setProbabilityMandatory(25);
 characteristics.setProbabilityOptional(25);
 characteristics.setProbabilityOr(25);
 characteristics.setProbabilityAlternative(25);
 characteristics.setMaxBranchingFactor(12);
 characteristics.setMaxSetChildren(6);

 // Max number of products of the feature model to be generated.
 characteristics.setMaxProducts(10000);

 // STEP 2: Generate the model with the specific characteristics
 AbstractFMGenerator generator = new FMGenerator();
 FAMAFeatureModel fm = (FAMAFeatureModel) generator.generateFM(characteristics);

 // OPTIONAL: Show detailed statistics of the feature model generated
 FMStatistics statistics = new FMStatistics(fm);
 System.out.println(statistics);

 // STEP 3: Save the model and the products
 FMWriter writer = new FMWriter();
 writer.saveFM(fm, "./model.xml");   // Other valid formats: .splx, .fm, .dot
 writer.saveProducts(generator.getPoducts(), "./products.csv");
}
```

Listing §A.3 shows how to use our evolutionary algorithm to generate a feature model maximizing the execution time invested by FaMa when retrieving the set of products represented by the model. As illustrated, the optimization criteria is provided to the generator as an input fitness function (shown in Listing §A.4).

Listing A.3: Random generation of a feature model maximizing the cross-tree constraint ratio

```
public static void main(String[] args) throws BettyException, Exception {

 // STEP 1: Specify the user's preferences for the generation (characteristics)
 GeneratorCharacteristics ch = new GeneratorCharacteristics();
 ch.setNumberOfFeatures(100);
```

```
  ch.setPercentageCTC(30);

  // STEP 2: Create a fitness function determining the optimization criteria (e.g. maximize
      CTCR)
  CTCRFitness fitnessFunction = new CTCRFitness();

  // STEP 3: Search for a feature model with the size constraints maximizing the CTC ratio.
  EvolutionaryFMGenerator generator = new EvolutionaryFMGenerator();
  generator.setFitnessFunction(fitnessFunction);
  FAMAFeatureModel fm = (FAMAFeatureModel) generator.generateFM(ch);

  // OPTIONAL: Show detailed statistics of the feature model generated
  FMStatistics statistics = new FMStatistics(fm);
  System.out.println(statistics);

  // STEP 4: Save the model
  FMWriter writer = new FMWriter();
  writer.saveFM(fm, "./model.xml");
  }
 }
```

Listing A.4: Fitness function measuring the analysis time of a feature model in FaMa

```
public class TimeFitness implements FitnessFunction {

 @Override
 public double fitness(FAMAFeatureModel fm) {
  double stime=System.currentTimeMillis();
  QuestionTrader qt = new QuestionTrader();
  ProductsQuestion pq = (ProductsQuestion) qt.createQuestion("Products");
  qt.setVariabilityModel(fm);
  qt.ask(pq);
  return System.currentTimeMillis()-stime;

 }
}
```

In this dissertation, we have mainly focused on optimization problems dealing with time and memory consumption. However, we may remark that our evolutionary algorithm is flexible enough to deal with other optimization criteria by simply defining adequate fitness functions. As an example, Listing §A.5 depicts a fitness function measuring the cross-tree constraint ratio of a feature model, i.e. ratio between the number of features involved in cross-tree constraints and the total number of features [21]. Using this function together with the code presented in Listing §A.3, we could easily generate feature models maximizing their cross-tree constraint ratio.

Listing A.5: Fitness function measuring the cross-tree constraints ratio of a feature model

```
public class CTCRFitness implements FitnessFunction {

 @Override
 public double fitness(FAMAFeatureModel fm) {
  FMStatistics statistics= new FMStatistics(fm);
  return statistics.getCTCR();
 }
}
```

A.4 Related tools

During our revision of related works, we found a good number of approaches using random feature models to test the performance of their tools (see Chapter §5). However, in most of the cases, these were generated using ad–hoc tools not publicly available. We found that only the SPLOT website [159] provides a standalone Java application for the generation of random feature models and their storage in Simple XML format. The tool, not extensible, receives several parameters constraining the size and properties of the feature model to be generated (e.g. number of features or percentage of mandatory features).

When compared to related works, BeTTy is the first extensible framework specifically design for functional and performance testing of feature model analysis tools. In addition to random generation of feature models, it also provides extra and novel features including the generation of products using metamorphic relations, guided generation of feature models using optimization criteria and support classes for benchmarking. Also, BeTTy support most popular formats for feature models and is distributed as a jar file facilitating its interoperability with other tools.

A.5 Summary

In this appendix, we have presented the BeTTy framework, an extensible and highly configurable tool supporting benchmarking and testing on the analyses of feature models. BeTTy integrates most of the contributions presented in this thesis. In fact, its main goal is to make our work accessible to the feature modelling community and encourage other researchers and practitioners to use and extend it. BeTTy uses some of the core components of FaMa and thus it is part of the FaMa ecosystem.

The framework is written in Java, distributed under LGPL v3 license and freely available at http://www.isa.us.es/betty.

Appendix B

Mutation operators

B.1 Traditional mutation operators

Operator	Description
AODS	Arithmetic Operator Deletion (Short-cut)
AODU	Arithmetic Operator Deletion (Unary)
AOIS	Arithmetic Operator Insertion (Short-cut)
AOIU	Arithmetic Operator Insertion (Unary)
AORB	Arithmetic Operator Replacement (Binary)
AORS	Arithmetic Operator Replacement (Short-cut)
ASRS	Assignment Operator Replacement (Short-cut)
COD	Conditional Operator Deletion
COI	Conditional Operator Insertion
COR	Conditional Operator Replacement
LOD	Logical Operator Deletion
LOI	Logical Operator Insertion
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
SOR	Shift Operator Replacement

Table B.1: *Traditional mutation operators.*

B.2 Class–level mutation operators

Language feature	Operator	Description
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	IPC	Explicit call of a parent's constructor deletion
	ISD	*super* keyword deletion
	ISI	*super* keyword insertion
Overloading	OAC	Arguments of overloading method call change
	OMD	Overloading method deletion
	OMR	Overloading method contents replace
Polimorphism	PCC	Cast type change
	PCD	Type cast operator deletion
	PCI	Type cast operator insertion
	PNC	*new* method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable variable
Java-specific features	JTI	*this* keyword insertion
	JTD	*this* keyword deletion
	JSI	*static* modifier insertion
	JSD	*static* modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common programming mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Acessor method change
	EMM	Modifier method change

Table B.2: *Class-level mutation operators.*

Appendix C
Mutation testing report

As shown in previous chapters, we made an extensive use of mutation testing to evaluate the effectiveness of our suite and our automated test data generator. During our work with mutation, we came up with some interesting findings and we had to overcome several difficulties that hindered the practical application of mutation to our domain. We felt that these findings could be of interest for the mutation community and thus we decided to shared this with them. This appendix reproduces the content of an experience report written by the authors and accepted for publication in the Information and Software Technology special issue on mutation testing [152]. This article was the result of a collaboration with Professor Robert M. Hierons who supervised our work during our research stay in his lab.

C.1 Introduction

Mutation testing [53] is a fault-based testing technique that measures the effectiveness of test cases. First, faults are introduced in a program creating a collection of faulty versions, called *mutants*. The mutants are created from the original program by applying changes to its source code (e.g. i + 1 ⇒ i - 1). Each change is determined by a *mutation operator*. Test cases are then used to check whether the mutants and the original program produce different responses, detecting the fault. The number of mutants detected provides a measure of the quality of the test suite called *mutation score*.

Mutation testing has traditionally been applied to procedural programs written in languages like Fortran or C. However, the increasing presence of Object–Oriented (OO) programs in industrial systems is progressively drawing the attention of mutation researchers toward this paradigm [82]. Contributions in this context mainly focus on the development of new tools and mutation operators (class-level operators) specifically designed to create faults involving typical OO features like inheritance or polymorphism. However, little research has been done to study the effectiveness and limitations of OO mutation in practice.

The high cost of mutation has traditionally hindered its application in empirical studies, including those involving OO systems. To make it affordable, some researchers apply it to basic or teaching programs [5, 89, 156, 157] rather than real-world applications. Others, save effort by mutating only a part of the system [5, 90, 95, 100, 156, 157] or using a subset of mutation operators [55, 72, 89, 100, 123]. In both cases, the representativeness of the results is therefore only partial. Beside this, related studies are in most cases reported by mutation experts rather than practitioners, whose experiences could probably provide more insights about the applicability of the approach. Finally, although some interesting results have been provided, these are often isolated since they are extracted from a single system. This leads authors to concede the need for more practical experiences that support, contrast or complement their results [90, 95, 157].

In this article, we report our experience using mutation testing to measure the effectiveness of an automated test data generator for the analysis of feature models. Our work contributes to the field of practical experimentation with mutation as follows:

- We conducted the experiments using FaMa [58, 168], an open source Java OO framework for the analysis of feature models. FaMa is a widely-used tool of significant size under continuous development. It is already integrated into the feature modelling tool MOSKitt [113] and is being integrated into the commercial tool pure::variants[†1] [131]. We are also aware of its use in different universities and laboratories for teaching and research purposes.

- We fully mutated three of the analysis components (so-called reasoners) integrated into FaMa using both traditional and class–level mutation operators for Java.

- We compared and contrasted our mutation results with the data from some motivating faults found in the literature and recent releases of FaMa and SPLOT [159], two real tools for the analysis of feature models. We are not aware of any other related work on mutation reporting results of real faults in OO programs.

[†1]This integration is being performed in the context of the DiVA European project (http://www.ict-diva.eu/)

- Equivalent mutants were detected and examined manually, rather than using partially-effective automated mechanisms [96, 123], providing helpful feedback about the detection effort and equivalence causes.

- To the best of our knowledge, this is the first work reporting the experience with mutation testing from a user perspective, leading to a number of lessons learned that could be helpful for both researchers and practitioners in the field.

The rest of this article is structured as follows. Section §C.2 provides an overview of the analysis of feature models and the FaMa Framework. The experimental setup used in our study and the analysis of results obtained are detailed in Sections §C.3 and §C.4 respectively. Section §C.5 compares and contrasts the mutation results with the data obtained when using our test data generator to detect some real faults in FaMa and SPLOT. In Section §C.6, we report our main findings in a number of lessons learned. The threats to validity of our study are presented in Section §C.7. Section §C.8 provides an overview of related studies. Finally, we summarize our main conclusions in Section §C.9.

C.2 FaMa framework

A *feature model* defines the valid combinations of features in a domain. These are widely used to represent the commonalities and variabilities within the products of a software product line [11]. The automated analysis of feature models is an active research area that deals with the computer–aided extraction of information from feature models. These analyses are generally performed in two steps. First, the model is translated into a specific logic representation (e.g. propositional formula). Then, off-the-shelf solvers are used to automatically perform a variety of *analysis operations* [21] on the logic representation of the model.

FaMa [58, 168] is an open source Java framework for the automated analysis of feature models. FaMa provides a number of extension points to plug in new analysis components called reasoners. A *reasoner* is an extension of the framework implementing one or more analysis operations using a specific paradigm. Currently, FaMa integrates ready-to-use reasoners enabling the analysis of feature models using SAT solvers (SAT), Binary Decision Diagrams solvers (BDD) and Constraint Programming solvers (CP). A simplified version of the FaMa architecture is shown in Figure §C.1.

FaMa is a widely-used tool of significant size under continuous development[†2]. It is already integrated into the feature modelling tool MOSKitt [113] and is being integrated into the commercial tool pure::variants [131]. We are also aware of its use in different universities and laboratories for teaching and research purposes. These reasons, coupled to our familiarity with the tool as leaders of the project, made us choose FaMa as a good candidate to be mutated. More specifically, we used three of the reasoners integrated into FaMa as subject tools for our study.

[†2]Lines of code: 22,723. Developers: 6. Total releases: 8. Frequency of releases: 5 months.

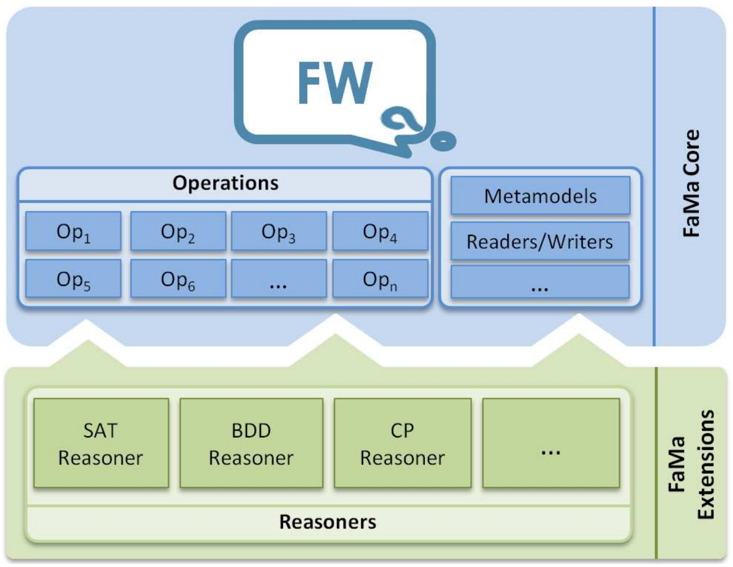


Figure C.1: *FaMa Architecture.*

C.3 Experimental setup

We next provide a full description of the mutation tool, mutation operators, subject programs, test data generator and experimental procedure used in the evaluation of our automated test data generator.

C.3.1 Mutation tool

For the generation of mutants, we used the MuClipse Eclipse plug-in v1.3 [157]. MuClipse is a visual Java tool for object-oriented mutation testing based on MuJava (Mutation System for Java) [95, 100]. We found this tool to provide helpful features for its use in our work, namely: i) wide range of mutation operators (including both traditional and class-level operators), ii) visual interface, especially useful for the examination of mutants and their statuses (i.e. alive or killed), iii) full integration with the development environment, and iv) support for the generation and execution of mutants in two clearly separated steps. Despite this, we found some limitations in the current version of the tool. Firstly, it does not support Java 1.5 code features. This forced us to make slight changes in the code, basically removing annotations and generic types when needed. Secondly, the tool did not support jar files. This was solved by manually placing the binary code of the application in the corresponding folder prior to the generation of mutants.

Regarding the execution of mutants, we found that MuClipse and other related tools were not sufficiently flexible, providing as results mainly mutation score and a list of alive and killed mutants. To address our needs, we developed a custom execution module providing some extra functionality, namely: i) custom results such as time required to kill each mutant and number of mutants generated by each operator, ii) results in Comma Separated Values (CSV) format for its later processing in spreadsheets, and iii) fine-grained filtering capabilities to specify which mutants should be considered or ignored during the execution. For each class under evaluation, our execution module works in two iterative steps. First, the binary file of the original class is replaced by the corresponding file of the current mutant. Then, test cases are run and results collected using the programmatic API of JUnit 4 [84].

C.3.2 Mutation operators

There are two types of mutation operators for OO systems: *traditional* and *class-level* operators. The former are those adapted from procedural languages such as C or Fortran. These mainly mutate traditional programming features such as algebraic or logical operators (e.g. i $\Rightarrow$ i++). The latter are specifically designed to introduce faults affecting OO features like inheritance or polymorphism (e.g. *super* keyword deletion). In our study, we applied all 15 traditional and 28 class-level operators for Java available in MuClipse, listed in Appendix §B. For details about these operators we refer the reader to [98, 99].

C.3.3 Experimental data

We chose three of the reasoners integrated into the FaMa Framework as subject tools for our experiments, namely: the Sat4jReasoner v0.9.2 (using satisfiability problems by means of Sat4j solver [138]), the JavaBDDReasoner v0.9.2 (using binary decision diagrams by means of JavaBDD solver [80]) and the JaCoPReasoner v0.8.3 (using constraint programming by means of JaCoP solver [79]). Each of these reasoners uses a different paradigm to perform the analyses and was coded by different developers, providing the required heterogeneity for the evaluation of our approach.

For each reasoner, we selected the classes extending the framework and implementing the analyses capabilities. Table §C.1 shows the number of classes selected from each reasoner together with the total number of lines of code[†3] (LoC) and the average number of methods and attributes per class. The size of the 27 subject classes included in our study ranged between 35 and 220 LoC. These metrics were computed automatically using the plug-in Metrics 1.3.6 for Eclipse [112].

Reasoner	**LoC**	**Classes**	**Av Methods**	**Av Attributes**
Sat4jReasoner	743	9	6.5	1.8
JavaBDDReasoner	625	9	6.3	2.1
JaCoPReasoner	686	9	6.3	2.3
Total	**2,054**	**27**	**6.4**	**2.1**

Table C.1: *Size statistics of the three subject reasoners.*

For each reasoner, the implementations of six different analysis operations were tested. A description of these operations is given in Table §C.2.

[†3]LoC is any line within the Java code that is not blank or a comment.

Operation	Description
VoidFM	Informs whether the input feature model is void or not (i.e. it represent no products)
ValidProduct	Informs whether the input product belongs to the set of products of a given model
Products	Returns the set of products of a feature model
#Products	Returns the number of products represented by a feature model
Variability	Returns the variability degree of a feature model
Commonality	Returns the percentage of products represented by the model in which a given feature appears

Table C.2: *Analysis operations tested.*

C.3.4 Test data generator

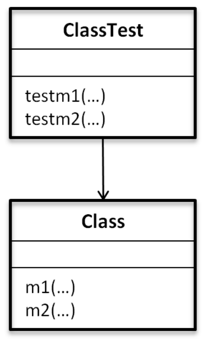
Test cases were automatically generated using the automated test data generator presented by the authors in [151]. This is designed to detect faults in the implementations of the analysis operations on feature models regardless of how these are implemented. Thus, it uses a black-box testing approach relying on the inputs and outputs of the analysis operations under test. Given an initial test case, the tool automatically generates random follow-up test cases using known relations between the input feature models and their expected outputs (so-called metamorphic relations). The test generation process was parameterised by a number of factors such as the number of features or the percentage of constraints of the input models to be generated. For each operation under test, we used our generator to generate and run test cases until a fault was found or a timeout was exceeded.

C.3.5 Mutants execution

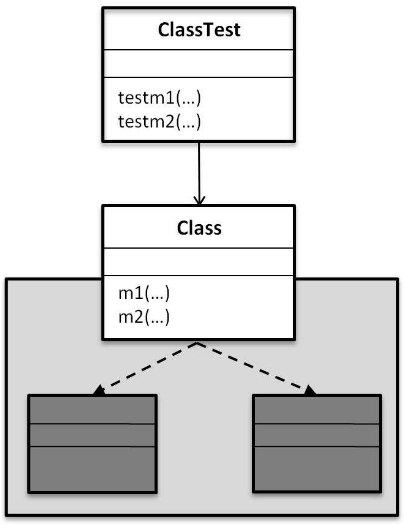
Traditionally, classes are considered the units of functionality during mutation testing in OO systems (Figure §C.2(a)). That is, each class is first mutated and then their mutants are executed against a test set. In our study, however, the nature of FaMa and our generator forced us to revise the notion of unit testing. The goal of our test data generator is to detect faults in the implementation of analysis operations regardless of how these are implemented. Thus, we consider a testing unit as a black-box component providing the functionality of an analysis operation (Figure §C.2(b)). No knowledge is assumed about the internal structure of the component, only about the interface of the operation that it implements. Note that this means that a testing unit could therefore be composed of more than one class.

As an example, consider the partial class diagram of Sat4jReasoner showed in Figure §C.3 (some associations are omitted for simplicity). As previously explained, we can only assume knowledge about the classes extending the public interfaces of the operations provided by the framework i.e. these are the only classes we generated test data for. Note that the functionality of the operations is not provided by a single class, but several of them that are often reused by different operations. For instance, class Reasoner participates in the implementation of all the operations under test. Once all classes in the reasoner were mutated, we evaluated the ability of our generator to kill the mutants in a three-step process as follows:

i. We grouped those classes providing the functionality of each analysis operations, i.e.



(a) *Class-level testing.*



(b) *Component-level testing.*

Figure C.2: *Class-level vs. component-level testing.*

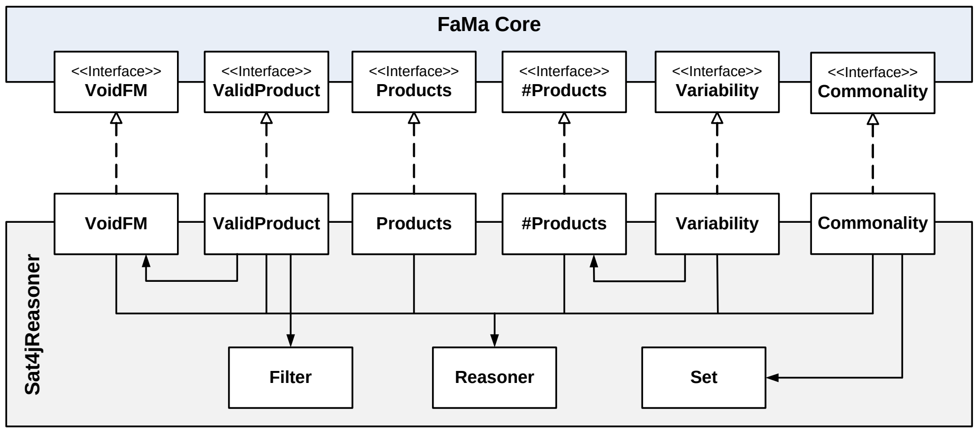


Figure C.3: *Partial class diagram of Sat4jReasoner.*

this could be considered a testing unit, a component. For instance, in Figure §C.3, classes {Variability+Reasoner+#Products} provide the functionality of the operation *Variability*.

ii. For each operation, we performed a set of executions trying to kill the mutants of the classes implementing the operation. On each execution, we mutated a single class and left the rest in the original form. For instance, let us use C_T to denote the set of mutants of a given class C. Tests in the operation *Variability* were evaluated by trying to kill mutants when testing the operation with the following combination of classes {Variability_T + Reasoner + #Products}, {Variability + Reasoner_T + #Products} and {Variability + Reasoner + $\text{\#Products}_T$}.

iii. To work out the results of each operation, we considered the results obtained on each of the executions associated to it.

The previous process raised new challenges to be solved during the execution and computation of results. To illustrate this, consider the simplified class diagram depicted in Figure §C.4. The Reasoner class is reused in the implementation of the operations *ValidProduct* and *Commonality*, i.e. it is shared by two different testing units. A label is inserted in those methods where a change was introduced generating a mutant, $M < i >$. We felt the need to distinguish clearly between the following terms:

- *Generated mutants*. These are the mutants generated in the source code, a total of 5 in Figure §C.4, i.e. ValidProduct_{M1}, Reasoner_{M2}, Reasoner_{M3}, Commonality_{M4} and Commonality_{M5}.
- *Executed mutants*. These are the mutants actually executed when considering the mutants in reusable classes. In the example, mutants in the reused class Reasoner (i.e. Reasoner_{M2} and Reasoner_{M3}) are executed against the test data of the operations *ValidProduct* and *Commonality* resulting in 7 mutants executed listed in the table of Figure §C.4.

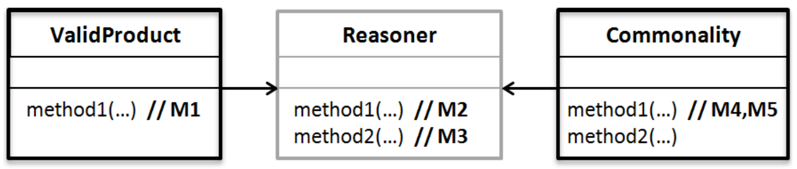
In addition, we identified two new mutant statuses to be considered when processing our results, namely:

- *Uncovered mutants*. We say a mutant in a reusable class is uncovered by a given operation if it cannot be exercised by that operation. For instance, let us suppose that class ValidProduct in Figure §C.4 does not use *method2* of the class *Reasoner*. We would then say that mutant Reasoner_{M3} is not covered by the operation *ValidProduct* (see executed mutant 3 in Figure §C.4). Uncovered mutants are a type of equivalent mutants and were manually identified and omitted for the computation of results on each operation.
- *Partially equivalent mutants*. We say a mutant in a reusable class is partially equivalent if it is equivalent for a subset of the operations using it (but not all of them). For instance, Reasoner_{M2} is alive when tested against the tests of the operation *ValidProduct* (i.e. $\{\text{ValidProduct} + \text{Reasoner}_{M2}\}$) but equivalent when tested with the test data of the operation *Commonality* (i.e. $\{\text{Commonality} + \text{Reasoner}_{M2}\}$). In contrast to conventional equivalent mutants, partially equivalent mutants were not excluded from our study. Instead, they were included and omitted only in those operations where they were identified as equivalent.

C.3.6 Experimental procedure

For the application of mutation testing in FaMa, we followed four steps, namely:

i. *Reasoners testing*. Prior to their analysis, we checked whether the original reasoner passed all the tests. A timeout of 600 seconds was used. As a result, we detected and fixed a defect in the JaCoPReasoner. We found this fault to be especially motivating since it was also present in the studied release of FaMa (see Section §C.5.2 for details).



Executed mutants	Status
1. ValidProduct$_{M1}$ + Reasoner	Killed
2. ValidProduct + Reasoner$_{M2}$	Alive
3. ValidProduct + Reasoner$_{M3}$	Uncovered
4. Commonality$_{M4}$ + Reasoner	Alive
5. Commonality$_{M5}$ + Reasoner	Killed
6. Commonality + Reasoner$_{M2}$	Equivalent
7. Commonality + Reasoner$_{M3}$	Killed

Figure C.4: *Example of mutation in reusable classes.*

ii. *Mutants generation.* We generated two sets of mutants by applying all traditional and class–level operators available in MuClipse (listed in Appendix §B).

iii. *Mutants execution.* For each executed mutant, we ran our test data generator until finding a test case that kills it or until a timeout of 600 seconds was exceeded. We chose this value for the timeout because it had proved to be effective in similar experiments with mutants and automated metamorphic testing [68].

iv. *Results processing.* We classified mutants into different categories. Generated mutants were classified into equivalent, not equivalent, undecided and discarded mutants. We marked as *undecided* those mutants whose equivalence we could not conclude in a reasonable time (more details in Section §C.4.3). *Discarded* mutants were those changing parts of the programs exercised by the tests (they were not uncovered) but that were of no interest for our study since they affected secondary functionalities not addressed by our test data generator (e.g. execution time measurement). Executed mutants were classified into killed, alive, uncovered and partially equivalent mutants. Finally, test data generation and execution results were processed for each operation.

All our experiments were performed on a laptop machine equipped with an Intel Pentium Dual CPU T2370@1.73GHz and 2048 MB of RAM memory running Windows Vista Business Edition and Java 1.6.0_05.

C.4 Analysis of results

In this section, we report the analysis of our results. We first outline the data obtained with both traditional and class-level mutation operators. Then, we describe our experience detecting equivalent mutants.

C.4.1 Analysis of results with traditional mutants

Table §C.3 depicts information about the mutants obtained when applying traditional mutation operators. We distinguish between "Generated Mutants" and "Executed Mutants", where a generated mutant is a mutated class and an executed mutant is a component (i.e. set of classes implementing an operation) that contains one mutated class (see Section §C.3.5 for details). In procedural programs, the number of generated mutants is typically large. For example, a simple Fortran program of 29 LoC computing the days between two dates results in 3,010 mutants [121]. In FaMa, however, the total number of generated mutants for the 27 subject classes was 749. This occurs because traditional mutants mainly modified features that the FaMa classes barely contained (e.g. arithmetic operators). Out of the total 749 mutants, 101 (13.4%) were identified as semantically equivalent. This percentage is within the boundaries reported in similar studies with procedural programs which suggest that between 5% and 15% of generated mutants are equivalent [54, 62, 120–123]. In addition to these, we also discarded 87 mutants (11.6%) affecting other aspects of the program not related to the analysis of feature models and therefore not addressed by our test data generator. These were mainly related to the computation of statistics (e.g. execution time) and exception handling.

Executed mutants only include killable mutants, i.e. not equivalent. The number of executed mutants was nearly twice the number of generated mutants. That means that many of the generated mutants were in reusable classes.

Operator	Generated Mutants			Executed Mutants			Score (%)
	Total	Equivalent	Discarded	Total	Alive	Killed	
AOIS	296	66	32	547	11	536	97.9
ROR	106	9	3	230	0	230	100
LOI	93	5	13	180	0	180	100
COI	87	5	3	180	0	180	100
AORB	68	7	24	58	0	58	100
AOIU	53	5	7	84	0	84	100
COD	19	0	5	21	0	21	100
AORS	18	0	0	57	0	57	100
COR	8	4	0	7	0	7	100
AODU	1	0	0	1	0	1	100
Total	**749**	**101**	**87**	**1,365**	**11**	**1,354**	**99.1**

Table C.3: *Traditional mutants for FaMa classes.*

Operators AODS, SOR, LOR, LOD and ASRS did not generate any mutants since they mutate language operators that the subject programs did not employ (e.g. unary logic). The operator AOIS produced the most mutants (39.5% of the total) followed by ROR (14.15%), LOI (12.4%) and COI(11.6%). The dominance of these four operators was also observed in related studies with Java programs [156, 157]. Regarding individual mutation scores, the operator AOIS, with a score of 97.9%, was the only one introducing faults not detected by our generator. Hence, this was the only operator providing helpful information to improve the quality of our tool.

Tables §C.4, §C.5 and §C.6 show the results obtained when using our test data generator to try to kill traditional mutants in the three subject reasoners. For each operation, the number of classes involved, number of executed mutants, test data generation results and mutation score are presented. Test data generation results include average and maximum time required to kill each mutant and average and maximum number of test cases generated before killing a mutant. These results provide us with quantitative data to measure and compare the effectiveness of our generator when killing mutants. Besides this, the average time and number of test cases generated before killing a mutant gives an idea of how difficult it was to detect faults in a certain operation. For instance, operations *Products*, *#Products*, *Variability* and *Commonality* showed a mutation score of 100% in all the reasoners with an average number of test cases required to kill each mutant under 2. This suggests that faults in these operations are easily killable. On the other hand, faults in the operations *VoidFM* and *ValidProduct* appeared to be more difficult to detect. We found that mutants on these operations required input models to have a very specific pattern in order to be revealed. As a consequence of this, the average time and number of test cases in these operations were noticeable higher than in the rest of analyses tested.

The maximum average time to kill a mutant was 7.4 seconds. In the worst case, our test data generator spent 566.5 seconds before finding a test case that killed the mutant (operation *VoidFM* in Table §7.3). In this time, 414 different test cases were generated and run. This gave us an idea of the minimum timeout that should be used when applying our approach in real scenarios.

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	Classes	Total	Alive	Av Time (s)	Max time (s)	Av TCs	Max TCs	
VoidFM	2	55	0	37.6	566.5	95.1	414	100
ValidProduct	5	109	3	4.3	88.6	12	305	97.2
Products	2	86	0	0.6	3.4	1.5	12	100
#Products	2	57	0	0.7	2.4	1.8	8	100
Variability	3	82	0	0.6	1.7	1.3	5	100
Commonality	5	109	0	0.6	3.8	1.5	13	100
Total	**19**	**498**	**3**	**7.4**	**566.5**	**18.9**	**414**	**99.3**

Table C.4: *Test data generation results using traditional operators in Sat4jReasoner.*

C.4.2 Analysis of results with class mutants

Table §C.7 shows information about the mutants obtained from class–level mutation operators. The total number of mutants is nearly 60% lower than the number of traditional mutants. This suggest that the OO features that these operators modify are less frequent than those features addressed by traditional operators such as arithmetic or relational operators. Out of the total 310 mutants, 141 (45.4%) were identified as semantically equivalent, far more than the 5% to 15% found in related studies with procedural programs. In certain cases, we could not conclude whether a mutant was equivalent or not. We marked these mutants, a total of 10 (3.2%), as "undecided". Finally, we discarded 10 (3.2%) mutants affecting secondary functionality of the programs not related to the analyses and not addressed by our test data generator.

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	Classes	Total	Alive	Av Time (s)	Max time (s)	Av TCs	Max TCs	
VoidFM	2	75	3	6.6	111.7	29.3	350	96
ValidProduct	5	129	5	1	34.6	3.8	207	96.1
Products	2	130	0	0.7	34.6	1.4	12	100
#Products	2	77	0	0.5	1.4	1.6	6	100
Variability	3	104	0	0.5	2.4	1.6	12	100
Commonality	5	131	0	0,5	3	1.5	16	100
Total	**19**	**646**	**8**	**1.6**	**111.7**	**6.5**	**350**	**98.7**

Table C.5: *Test data generation results using traditional operators in JavaBDDReasoner.*

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	Classes	Total	Alive	Av Time (s)	Max time (s)	Av TCs	Max TCs	
VoidFM	2	8	0	1.5	8.3	11.3	83	100
ValidProduct	5	61	0	0.7	1.2	1.3	5	100
Products	2	37	0	0.5	0.7	1	1	100
#Products	2	13	0	0.5	0.7	1	1	100
Variability	3	36	0	0.5	0.7	1	1	100
Commonality	5	66	0	0.5	0.7	1.1	3	100
Total	**19**	**221**	**0**	**0.7**	**8.3**	**2.8**	**83**	**100**

Table C.6: *Test data generation results using traditional operators in JaCoPReasoner.*

Class–level operators IHD, IOR, ISI, ISD, IPC, PMD, PPD, PCC, OMR, OMD, OAN, JSD, JID, EOA and EOC did not generate any mutants and therefore they are not showed in the table. We found that these operators mainly mutate inheritance and polymorphism features that the subject classes did not use. The operator PCI produced the most mutants with nearly 40% of the total. This operator inserts type casting to variables (e.g. feature ⇒ (GenericFeature) feature) and therefore is likely to be applied more frequently than other operators. Operators PCI and PNC generated the highest percentages of equivalent mutants with 78.8% and 57.8% respectively.

Operators PCI, IOD, EAM, JDC, PNC, PRV, JTI, JTD, IHI and IOP got a mutation score of 100%. This suggests that the mutants generated by these operators in FaMa can be easily killed with the same test cases generated for traditional mutants. These operators were therefore useless in terms of providing information to improve the quality of our tests. On the other hand, operators JSI and EMM got a mutation score of 0% providing us with information to improve our tool. For instance, we learned that using sequences of calls to several instances of the same class (instead of one instance per test case) would allow us to kill those mutants generated by the operator JSI (*static* modifier insertion), and related faults, making our test suite stronger.

Tables §C.8, §C.9 and §C.10 show the results obtained when using our test data generator

Operator	Generated Mutants				Executed Mutants			Score (%)
	Total	Equivalent	Undecided	Discarded	Total	Alive	Killed	
PCI	118	93	0	0	108	0	108	100
IOD	55	13	0	0	60	0	60	100
JSI	47	14	0	0	52	52	0	0
EAM	20	2	0	6	44	0	44	100
JDC	19	11	0	0	14	0	14	100
PNC	12	2	6	0	12	0	12	100
EMM	10	0	4	0	12	12	0	0
PRV	9	0	0	2	26	0	26	100
JTI	6	0	0	1	5	0	5	100
PCD	5	5	0	0	0	0	0	N/A
JTD	4	0	0	1	3	0	3	100
IHI	3	1	0	0	3	0	3	100
IOP	2	0	0	0	3	0	3	100
Total	**310**	**141**	**10**	**10**	**342**	**64**	**278**	**81.2**

Table C.7: *Class mutants for FaMa classes.*

to try to kill the class mutants. The mutation scores on each solver ranged from 69.6% to 91.9%. This means that our generator, despite not being specifically designed to detect OO faults, was able to detect a majority of the class mutants introduced in FaMa. The average time to kill each mutant was between 0.7 and 1.5 seconds. Similarly, the average number of test cases generated before killing each mutant was between 4.4 and 4.7. These results are better than the ones obtained when using our generator with traditional mutants. This reveals that class mutants that were killed, were also easier to kill than traditional ones.

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	Classes	Total	Alive	Av Time (s)	Max time (s)	Av TCs	Max TCs	
VoidFM	2	10	3	5,7	37	21	141	70
ValidProduct	5	31	9	1.2	5.7	3	20	71
Products	2	12	3	0.7	0.9	1	1	75
#Products	2	10	3	0.6	0.7	1	1	70
Variability	3	12	4	0.6	0.9	1.2	2	66.7
Commonality	5	27	9	0.5	0.7	1	1	66.7
Total	**19**	**102**	**31**	**1,5**	**37**	**4.7**	**141**	**69.6**

Table C.8: *Test data generation results using class-level operators in Sat4jReasoner.*

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	
VoidFM	2	7	1	1.4	5.1	19.8	112	85.7
ValidProduct	5	26	7	0.5	1.1	1.4	4	73.1
Products	2	7	1	0.6	1	1.7	3	85.7
#Products	2	7	1	0.7	1	2.5	4	85.7
Variability	3	9	2	0.5	1	1.6	4	77.8
Commonality	5	24	8	0.5	0.9	1.4	3	66.7
Total	**19**	**80**	**20**	**0.7**	**5.1**	**4.7**	**112**	**75**

Table C.9: *Test data generation results using class-level operators in JavaBDDReasoner.*

Operations		Executed Mutants		Test Data Generation				Score (%)
Name	**Classes**	**Total**	**Alive**	**Av Time (s)**	**Max time (s)**	**Av TCs**	**Max TCs**	
VoidFM	2	21	0	3	31.2	20.5	226	100
ValidProduct	5	35	4	0.6	1.5	1.5	9	88.6
Products	2	22	1	0.6	1.3	1	1	95.5
#Products	2	21	1	0.6	1	1.3	5	95.2
Variability	3	24	2	0.7	1.1	1.3	5	91.7
Commonality	5	37	5	0.6	1.4	1.1	4	86.5
Total	**19**	**160**	**13**	**1**	**31.2**	**4.4**	**226**	**91.9**

Table C.10: *Test data generation results using class-level operators in JaCoPReasoner.*

C.4.3 Equivalent mutants

The detection of equivalent mutants was performed by hand investing no more than 15 minutes per mutant in the worst case. If this time was exceeded without reaching a conclusion the mutants was marked as “undecided". In many cases, equivalent mutants were caused by similar structures in different classes. Once an equivalent mutant was detected, we found that detection of similar equivalent mutants was easier and faster. This suggests that the effort required to identify equivalent mutants is affected by the commonalities of the subject classes.

Regarding hardness, we found it especially difficult to determine the equivalence in those mutants replacing methods calls. In our study, these were generated by the class–level operators EMM and PNC. As an example, consider the following mutant generated by the operator PNC in Sat4jReasoner: new DimacsReader(solver) $\Rightarrow$ new CardDimacsReader(solver). The mutant changes the default reader used by Sat4j (off-the-shelf solver used by Sat4jReasoner) to process input files. Determining whether the functionality of the new reader is equivalent to the original one for any input problem was not trivial, even when checking the available documentation of Sat4j. This and other related mutants were therefore marked as “undecided".

C.5 Real faults

For a further evaluation of our approach, we checked the effectiveness of our tool in detecting real faults. This allowed us to study the representativeness of mutants when compared to faults identified in real scenarios. In particular, we first studied a motivating fault found in the literature. Then, we used our generator to test recent releases of two tools, FaMa and SPLOT, detecting two defects in each of them. These results are next reported.

C.5.1 A motivating fault found in the literature

Consider the work of Batory in SPLC'05 [11], one of the seminal papers in the community of automated analysis of feature models. The paper included a bug (later fixed[†4]) in the mapping of feature models to propositional formulas. Although this is not a real fault (i.e. it was not found in a tool), it represents a real type of fault likely to appear in real scenarios. This fault is motivational for two main reasons, namely: i) it affects all the analysis operations using the wrong mapping and ii) it is difficult to detect since it can only be revealed with input feature models containing a very specific pattern. For more details about this fault we refer the reader to [151]. We implemented this wrong mapping into a mock reasoner for FaMa using JavaBDD and tried to detect the fault using our test data generator.

Table §C.11 depicts the results of the evaluation. The testing procedure was similar to the one used with mutation testing. A maximum timeout of 600 seconds was used. All the results are the average of 10 executions. The fault was detected in all operations remaining latent in 50% of the tests performed in the *ValidProduct* operation. When examining the data, we concluded that this was due to the basic strategies used in our test data generator for the selection of inputs products for this operation. We presume that using more complex heuristics for this purpose would improve the results.

Operation	**Av Time (s)**	**Max Time (s)**	**Av TCs**	**Max TCs**	**Score (%)**
VoidFM	78.2	229.1	515.8	905	100
ValidProduct	38.4	43.7	268.4	322	50
Products	1.1	2.9	5.7	19	100
#Products	1.0	2.7	5.4	16	100
Variability	1.2	2.1	6.4	13	100
Commonality	1.4	3	7.8	20	100
Total	**20.2**	**229.1**	**134.9**	**905**	**91.6**

Table C.11: *Test data generation results using a motivating fault reported in the literature (averages of 10 executions).*

The average time to detect the fault (20.2 s) was far higher than the average times obtained when killing mutants, ranging between 0.7 and 7.4 seconds. Similarly, the average number of test cases generated before detecting the fault (134.9) exceeded the averages obtained in the

[†4]ftp://ftp.cs.utexas.edu/pub/predator/splc05.pdf

execution of mutants ranging between 4.4 and 18.9. Thus, our results reveal that detecting the fault in our mock tool was harder than killing the FaMa mutants.

C.5.2 FaMa Framework

We also evaluated our tool by trying to detect faults in a recent release of the FaMa Framework, *FaMa v1.0 alpha*. A timeout of 600 seconds was used for all the operations since we did not know *a priori* the existence of faults. Tests revealed two defects. The first one, also detected during our experimental work with mutation, was caused by an unexpected behaviour of JaCoP solver when dealing with certain heuristics and void models in the operation *Products*. In these cases, the solver did not instantiate an array of variables raising a null pointer exception. The second fault affected the operations *ValidProduct* and *Commonality* in Sat4jReasoner. The source of the problem was a bug in the creation of propositional clauses in the so-called staged configurations, a new feature of the tool.

Table §C.12 shows the results of our tests in FaMa. The results are the average of the data obtained in 10 executions. As illustrated, the defect in Sat4jReasoner was easy to detect in almost all cases with just one test case. On the other hand, the fault in JaCoPReasoner was harder to detect with a detection time of 160.9 seconds and 391.1 test cases generated on average. These results are again much higher than the averages obtained with mutation suggesting that this fault was harder to detect than mutants.

Operation	Av Time (s)	Max Time (s)	Av TCs	Max TCs	Score (%)
JaCoPReasoner - Products	160.9	214.9	391.1	535	100
Sat4jReasoner - ValidProduct	0.9	1.2	1	1	100
Sat4jReasoner - Commonality	0.9	1.3	1.1	2	100
Total	**54.2**	**214.9**	**131.1**	**535**	**100**

Table C.12: *Test data generation results in FaMa 1.0 alpha (averages of 10 executions).*

C.5.3 SPLOT

Software Product Lines On-line Tools (SPLOT) [159] is a Web portal providing a complete set of tools for on-line editing, analysis and storage of feature models. It supports a number of analyses on cardinality-based feature models using propositional logic by means of the Sat4j and JavaBDD solvers. The authors of SPLOT kindly sent us a standalone version[†5] of their system to evaluate our automated test data generator. In particular, we tested the operations *VoidFM*, *#Products* and *DeadFeatures* in SPLOT. As with FaMa, we used a timeout of 600 seconds and tested each operation 10 times to get averages. Tests revealed two defects in all the executions. The first one affected all operations on the SAT-based reasoner. With certain void models, the reasoner raised an exception (*org.sat4j.specs.ContradictionException*) and no result was returned. The second bug was related with cardinalities in the BDD-based tool.

[†5]SPLOT does not use a version naming system. We tested the tool as it was in February 2010.

We found that the reasoner was not able to process cardinalities other than [1,1] and [1,*]. As a consequence of this, input models including or-relationships specified as [1,n] (n being the number of subfeatures) caused a failure in all the operations tested.

Table §C.13 depicts the results of our tests in SPLOT. Notice that the *DeadFeatures* operation was only implemented in the SAT-based reasoner. Detection times were even lower than those found with traditional mutants in FaMa suggesting that the bugs were easier to detect than mutants. We may remark, however, that SPLOT was much faster than FaMa in executing test cases and therefore time does not provide a fair comparison. From the average number of test cases, however, we found that the fault in the SAT-based reasoner required the generation of more test cases (between 26.7 and 38.3 on average) than FaMa mutants suggesting that this fault was slightly harder to detect than mutants in general. The fault in the BDD-based reasoner, on the other hand, was trivially detected in all cases.

Operation	Av Time (s)	Max Time (s)	Av TCs	Max TCs	Score (%)
Sat4jReasoner - VoidFM	0.7	1.3	26.7	66	100
Sat4jReasoner - #Products	1	2	26.1	66	100
Sat4jReasoner - DeadFeatures	0.9	2.2	38.3	134	100
JavaBDDReasoner - VoidFM	0.4	0.5	1.5	2	100
JavaBDDReasoner - #Products	0.4	0.5	1.9	5	100
Total	**0.7**	**2.2**	**18.9**	**134**	**100**

Table C.13: *Test data generation results in SPLOT (averages of 10 executions).*

Faults detected in the standalone version of the tool were also observed in the online version of SPLOT. We may remark that authors confirmed the results and told us they were aware of these limitations.

C.6 Discussion and lessons learned

Our results can be summarized in the following lessons learned:

The number of mutants was lower than in procedural programs. The number of mutants generated by traditional mutation operators in FaMa was much lower than the number of mutants generated in smaller procedural programs [54, 62, 120–122]. This was also observed in related studies with OO systems [90, 95, 156, 157]. Like FaMa, the OO systems used in related studies do not perform many arithmetic or logical operations keeping the number of traditional mutants lower than in procedural programs. This suggests that mutation testing could be affordable when applied to OO applications with a reduced number of these features. This suggests that it would be useful to have a prediction model (e.g. equation) that practitioners could use to estimate the cost of mutation according to the characteristics of their programs. While some progress has been made in predicting the number of mutants in procedural languages [121], this seems to remain a challenge in the context of OO programs in which only some preliminary analysis has been proposed. [97].

The number of class mutants was lower than the number of traditional mutants. The number of mutants generated by class–level mutation operators was lower (about 60%) than traditional mutants. This trend was also observed in related studies with OO systems [55, 90, 95, 96, 100, 123]. This suggests that the OO features mutated by class-level operators occur less frequently than those addressed by traditional ones. It is worth remarking that even if we consider both traditional and class mutants in our study, there were still far fewer mutants than have been reported with smaller procedural programs. This also supports the applicability of mutation testing in OO systems.

Class–level operators generated more equivalent mutants than traditional ones. Our results show that the percentage of equivalent mutants generated by class–level operators (i.e. 45.4%) is much higher than the one generated with traditional operators (i.e. 13.4%). This result is also higher than the percentages reported in similar studies with procedural programs suggesting that between 5% and 15% of generated mutants are equivalent. This makes class-level mutation operators less attractive for experimentation since they generate fewer mutants than traditional operators and a larger percentage of equivalent ones reducing the total portion of useful (i.e. killable) mutants. Having said this, we may remark that we found contradictory results in the literature. On the one hand, studies conducted by Kim *et al.* [90] and Ma *et al.* [95] reported percentages of equivalent mutants of 4.9% (23 out of 466) and 0.4% (3 out of 691) respectively, much less than the percentage found in our study. On the other hand, a more recent experiment conducted by Ma *et al.* [96] revealed that at least 86% of the class mutants were equivalent, far more than the percentage found in our evaluation. We may mention that their experiment was performed on six open source programs with a total of 49,071 mutants studied which makes their results stronger than previous findings. When studying this work, we found that 9 of the mutation operators that generated higher percentages of equivalent mutants in their study (73.1% on average) did not generate any mutants in our work. We think this explains why the percentage of equivalent mutants in our work is lower supporting their result. Nevertheless, the heterogeneous data found in the literature suggest that a more extensive experiment using a wider number of subject programs would be highly desirable to shed light on this issue.

A majority of the class mutants were killed with the same kind of test cases designed for traditional mutants. Over 80% of the class mutants were killed with the same type of test data generated to kill traditional mutants. This was also observed by Ma et al. [95] in the BCEL system who found that more than 50% of the class mutants were killed with the same test cases used to kill traditional mutants. A further result in our study is that class mutants that were killed, were also easier to kill than traditional ones in terms of time invested by our generator to detect them. These results reveal that most class–level operators generated easily killable faults and therefore were not helpful in providing information to improve the quality of our test data generator. Only a small subset of these, JSI and EMM in our study, showed to be effective in providing feedback for the refinement of our tests.

Real faults were harder to detect than artificial ones. Three out of the five real faults studied in our work had average detection times or average number of test cases generated significantly higher than those of traditional and class mutants. Operation *ValidProduct*, for instance, got the lowest score (50%) when studying the motivating fault found in the literature (see Section §C.5.1). This suggests that real faults (based in our study) were, on average, harder to detect than mutants. We may remark, however, that this was not the case for all mutants. Hence, the highest detection time (566.5 seconds) was obtained when evaluation a mutant in the operation *VoidFM* of Sat4jReasoner (see Table §7.3). This means that mutants were able to represent faults as hard to detect as real ones. More importantly, the mutation scores obtained when studying real faults were similar (100% in the case of tests in FaMa 1.0 alpha and SPLOT) to those obtained with mutation in FaMa reasoners. This supports the *"competent programmer assumption"* that underlies the theory of mutation testing. That is, the assumption that faults made by programmers will be detected by those test suite able to kill mutants.

Relative cost of detecting equivalence mutants. The detection of equivalent mutants was easier and faster as we progressed and we found equivalent mutants similar to those previously identified. This means that the effort required to detect equivalent mutants was influenced by the similarities found in the subject programs. In our study, commonalities were frequent since the three reasoners implemented common interfaces provided by the framework. It would be interesting to know to what extent this happens in other tools and what impact this has on the cost of mutation testing.

Mutation testing on non–trivial testing units. Mutation testing in OO systems is traditionally applied at the class level, i.e. each single class is considered a unit of functionality. In our study, however, we consider a testing unit as a black–box component providing the functionality of an analysis operation. No knowledge is assumed about the internal structure of the component, only about the interface that it implements. Although possible, we concluded that the application of mutation testing in this context is hard and time–consuming. A lot of manual work was required to mutate each class individually, classify its mutants and compute the results of each operation afterward. This was even more challenging when considering reusable classes participating in the implementation of different operations. This introduced two new statuses for mutants, *uncovered* and *partially equivalent* mutants, and this made the processing of results even more difficult. Emerging works studying the application of mutation testing at the system and functional levels are a promising starting point to address this issue [104].

Limitations in current mutation tools. Mutation tools for Java include ExMan [31], Javalanche [72], JavaMut [43], Jester [81], Jumble [83], MuJava [95, 100] and MuClipse [157]. At the time of writing this article, we found that MuJava and its plug-in for Eclipse, MuClipse, were the only publicly available tools that provided support for class–level mutation operators in Java. When used in our study, we found this and other related tools had several practical limitations. In the following, we summarize those features that we missed during our study as well as those that we found more useful and that we consider should be part of any successful mutation tool:

- *Highly automated.* From a user perspective, we missed a major degree of automation in current tools particularly when concerning the computation of results. In addition to basic results such as the mutation score and list of alive and killed mutants, detailed statistics for each operator would be highly desirable. A deep analysis of these statistics would help users in taking decisions like detecting candidate operators to be omitted or selected in future evaluation or refinements of a test suite.
- *Flexible.* Tools should be able to work with real-world applications. Note that this not only means support for a large number of files or lines of code but also support for conventional library formats (e.g. jar files) as well as different versions of Java, features that we missed in our study.

- *Configurable*. Mutation tools should provide configuration utilities and extension points. Adding new mutation operators and result formats should be possible. Filters are also a helpful feature that we missed in our work. During generation, it would have been useful to have filters to specify fragments of code not to be mutated as this would have saved us the time of discarding mutants manually. During execution, our custom filter assisted us in selecting the set of mutants to be considered or omitted. This was especially useful when studying mutant equivalence.

- *Usable*. Mutation tools must be intuitive and easy to use. Visual interfaces (e.g. for examining mutants), configuration wizards and integration with the development environment proved to be helpful features in our study.

- *Automated detection of equivalent mutants*. Although the detection of equivalent mutants is an undecidable problem, some techniques have been proposed to automate it, at least partially [75, 120, 122, 123]. The development, improvement and integration of these techniques in mutation tools is undoubtedly one the main points to make mutation testing affordable in real scenarios.

C.7 Threats to validity

Our mutation results apply only to three of the reasoners integrated into FaMa framework and therefore may not extrapolate to other programs. Similarly, the number of real faults studied was not large enough to allow us to draw general conclusions. We may remark, however, that 4 out of the 5 real faults studied were found in current releases of two real tools which make them especially motivating for our study. Furthermore, we emphasize that our results may be helpful in supporting and complementing the results obtained in similar studies with OO applications.

The detection of equivalent mutants, an undecidable problem in general, was performed by hand resulting in a tedious and error-prone task. Thus, we must concede a small margin of error in the data regarding equivalence. We remark, however, that these results were taken from three different reasoners providing a fair confidence in the validity of the average data.

C.8 Related studies

Several researchers have reported the result of experimentation with mutation in OO systems. Kim *et al.* [89] were the first to introduce *Class Mutation* as a means to introduce faults targeting OO specific features in Java. In their work, the authors presented three mutation operators and applied them to three simple classes (Queue, Dequeue and PriorityQueue) resulting in 23 mutants studied. Later, in [90], the authors presented a larger empirical study using mutation testing to evaluate the effectiveness of three different OO testing methods. They applied both traditional (unspecified number) and 15 class–level mutation operators. As subject program, they used a subset of five classes (690 LoC) from the experimental system Product Starter Kit for Java of IBM. Results included the mutation scores for each testing technique under evaluation together with killed rates for each class–level operator. As a final remark, the authors concluded that their work was not extensive and further experimental work would be useful.

Ma *et al.* [100] presented a method to reduce the execution cost of mutation testing. The authors used MuJava to mutate the 266 classes of BCEL, a popular byte code engineering library, and collected data on the number of mutants generated by 24 class–level mutation operators. They also selected seven BCEL classes and studied them in depth comparing the performance of different techniques for the generation and execution of mutants. In a later work [95], the authors conducted two empirical studies to evaluate the usefulness of class–level mutation operators. They applied 5 traditional and 23 class–level mutants to the BCEL system collecting generation statistics for each operator. They also reported some conclusions based on a further study on 11 BCEL classes including killing rates and number of equivalent mutants. Again, the authors conceded that the study was conducted on one sample program and thus the result may not be representative.

Smith *et al.* [156] conducted an empirical study using the MuClipse mutation tool in the back–end of a small Java web–based application, iTrust, developed for academic purposes. In their study, the authors studied the behaviour of mutation operators and tried to identify patterns in the implementation code that remained untested. They applied all 15 traditional and 28 class–level operators available in MuClipse to three of the classes of the iTrust system. Detailed statistics for each operator were reported and analysed. In a later work [157], the authors extended their empirical study by mutating three of the classes of the open source library HtmlParser 1.6.

Offutt *et al.* [123] studied the number of mutants generated when applying mutation to the 866 classes of six open source Java projects. They applied 29 class–level operators using MuJava mutation tool. Generation statistics were presented for each operator including approximate percentage of equivalent mutants calculated using an automated technique. No results about execution of mutants were provided. Later, in [96], Ma *et al.* extended the work by studying the class-level operators that generated less mutants and concluded that some of them could be eliminated.

Alexander *et al.* [5] proposed an object mutation approach along with a set of mutation operators for inserting faults into objects implementing well–known Java interfaces (e.g. Iterator). To show the feasibility of their approach, they applied their mutation operators to five open source and classroom programs. They mutated, however, only a few locations in one class for each program resulting in 128 mutants studied.

Grün *et al* [72] study the impact of equivalent mutants. For their study, the authors mutated the JAXEN XPATH query engine (12,449 LoC), a popular open source project, resulting in 9,819 mutants. The authors used a custom mutation tool for Java, Javalanche, implementing a small set of sufficient traditional operators as proposed by Offutt [121] and later adapted by Andrews et al. [7]. They selected a subset of 20 mutants and studied them in depth reporting result about the number of equivalent mutants as well as the causes that made them to be equivalent.

Derezinska *et al.* [55] studied the application of mutation testing in C# programs. In their work, the authors applied 5 out of 40 mutation operators proposed for C# to nine available programs with a total size of 298,460 LoC (including comments). Mutation was performed using their tool CREAM (CREAtor of Mutants). As results, generation and execution statistics were given for each mutation operator.

When compared to previous studies, our work contributes to the field of practical experimentation with mutation in different ways. We used a real–world application as subject tool for the mutation instead of using sample or teaching programs [5, 89, 156, 157]. We fully mutated three of the reasoners integrated into FaMa rather than selecting a subset of classes from them [5, 90, 95, 100, 156, 157]. Similarly, we selected a complete set of traditional and class-level operators for Java instead of using a subset of these [55, 72, 89, 100, 123]. Equivalent mutants were detected and examined manually, rather than using partially effective automated mechanisms [96, 123], providing helpful feedback about the detection effort and equivalence causes. In addition to this, we complement our mutation results with the results of some real faults found in the literature and two real tools for the analysis of feature models. Our results include the time invested by our generator to detect each fault which provides interesting information to compare mutants and real bugs. We are not aware of any other work reporting results of real faults in OO programs in the context of mutation. Similarly, we are not aware of any other experience report providing results of mutation applied at a functional level considering testing unit as black-boxes instead of a single class. Finally, to the best of our knowledge, this is the first work reporting the experience with mutation testing from a user perspective, leading to a number of lessons learned helpful for both researcher and practitioner in the field following our steps.

C.9 Conclusions

This article reports our experience gained from using mutation testing to measure the effectiveness of an automated test data generator for the automated analysis of feature models. We applied both traditional and class–level mutation operators to three of the reasoners integrated into FaMa, an open source Java framework currently used for teaching, research and commercial purposes. We also compared and contrasted our results with the data obtained from some motivating faults found in the literature and recent releases of two real tools, FaMa and SPLOT. A key contribution of our study is that it is reported from a user perspective focusing on how mutation can help when evaluating a test mechanism. Our results are summarized in a number of lessons learned emphasizing important issues concerning the effectiveness and limitations of OO mutation in practice. These may be the seed for further research in the field.

Overall, the mutation scores obtained in our study were supported by the results obtained when detecting real faults in FaMa and SPLOT. This shows the effectiveness of mutation testing in measuring the ability of our generator in detecting faults. Also, the number of mutants generated in our subject classes was much lower than the number of mutants generated in procedural programs supporting the applicability of mutation testing in OO systems. Regarding drawbacks, we found several practical limitations in the current tool support for mutation that hindered our work. We also found that development of techniques and tools for the application of mutation at a system level is a challenging open issue in the mutation testing community.

Appendix D

Statistical analysis data

This appendix describes the statistical analysis of the experimental data obtained during the evaluation of our evolutionary algorithm presented in Chapter §8. The goal of statistical analysis is to provide formal and quantitative evidences showing that the algorithm works and that the results were not obtained by mere chance. The statistical analysis of the data was performed using the SPSS 17 statistical package [77].

Statistical analysis is usually performed by formulating two contrary hypothesis. The first hypothesis is referred to as *null hypothesis* and assume that the algorithm has no impact at all on the goodness of the results obtained, i.e. there is no difference between our algorithm and random search. Opposite to the null hypothesis, an *alternative hypothesis* is formulated, stating that the algorithm has a significant effect in the quality of the results obtained. Statistical tests provide a probability (named *p-value*) ranging in [0,1]. The lower the p-value of a test is, the more likely that the null hypothesis is false and the alternative hypothesis is true, i.e. the algorithm works. Alternatively, high p-values indicates more chances of the null hypothesis being true i.e. the algorithm does not work. Researchers have established by convention that p-values under 0.05 or 0.01 are statistically significant and are sufficient to reject the null hypothesis, i.e. prove that the algorithm is actually working.

The techniques used to perform the statistical analysis and obtain the p-values depend on whether the data follow a normal frequency distribution or not. The former assumes that data has come from a type of probability distribution and makes inferences about the parameters. The latter makes no assumptions at all. After some preliminary tests (Kolmogorov-Smirnov and Shapiro-Wilk tests) we concluded that our data did not follow a normal distribution and thus our tests required the use of so-called non–parametric techniques. In particular, we applied the Mann-Withney U non–parametric test to the experimental results obtained with our evolutionary algorithm and random search. Tables §D.1 and §D.2 show the results of these tests in SPSS for the experiments #1 and #2 respectively. For each number of features and percentage of cross-tree constraints, the values of the test are provided. The p-values are those labelled in the tables as "Asymp. Sig. (2-tailed)". As illustrated, tests rejected null hypotheses with extremely low p-values (zero in most of the cases) for nearly all experimental configurations of both experiments. This, coupled to the results shown in Chapter §8, clearly shows the great superiority of our algorithm when compared to random search. Only when the percentage of cross tree constraints (CTC) was 10% in Experiment #1, statistical test accepted some null hypotheses. As explained in Section §8.4, this problem is due to the small complexity of the analysis on those models. This problem makes our fitness landscape extremely flat, with scarce and disperse points of high fitness, where a random algorithm can find solutions nearly as good as those found by our evolutionary algorithm.

We may remark that the statistical results reported in this appendix were obtained with the help of J. A. Parejo, member of our research group and experienced researcher in the field of metaheuristics. Together, we plan to prepare a journal article to present our evolutionary algorithm and the associated experimental evaluation.

For more details about statistical tests and their meaning we refer the reader to [175].

Features	CTC	Test results	
200	10	Mann-Whitney U/Wilcoxon W	103/313
		Z	-2.631
		Asymp. Sig. (2-tailed)	0.009
	20	Mann-Whitney U/Wilcoxon W	26.5/236.5
		Z	-4.704
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	22/232
		Z	-4.834
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	42.5/252.5
		Z	-4.268
		Asymp. Sig. (2-tailed)	0
400	10	Mann-Whitney U/Wilcoxon W	73.5/283.5
		Z	-3.425
		Asymp. Sig. (2-tailed)	0.001
	20	Mann-Whitney U/Wilcoxon W	36/246
		Z	-4.44
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	33/243
		Z	-4.522
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	8/218
		Z	-5.21
		Asymp. Sig. (2-tailed)	0
600	10	Mann-Whitney U/Wilcoxon W	169.5/379.5
		Z	-0.826
		Asymp. Sig. (2-tailed)	0.409
	20	Mann-Whitney U/Wilcoxon W	32.5/242.5
		Z	-4.532
		Asymp. Sig. (2-tailed)	0

Features	CTC	Test results	
	30	Mann-Whitney U/Wilcoxon W	10.5/220.5
		Z	-5.131
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	4.5/214.5
		Z	-5.303
		Asymp. Sig. (2-tailed)	0
800	10	Mann-Whitney U/Wilcoxon W	141.5/351.5
		Z	-1.584
		Asymp. Sig. (2-tailed)	0.113
	20	Mann-Whitney U/Wilcoxon W	69/279
		Z	-3.545
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	9/219
		Z	-5.172
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	18/228
		Z	-4.928
		Asymp. Sig. (2-tailed)	0
1000	10	Mann-Whitney U/Wilcoxon W	93/303
		Z	-2.896
		Asymp. Sig. (2-tailed)	0.004
	20	Mann-Whitney U/Wilcoxon W	42.5/252.5
		Z	-4.261
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	51.5/261.5
		Z	-4.021
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	8.5/218.5
		Z	-5.192
		Asymp. Sig. (2-tailed)	0
(a). Not corrected for ties.			
(b). Grouping Variable: Algorithm (random or evolutionary)			

Table D.1: Experiment #1 Test Statistics

Features	CTC	Test results	
50	10	Mann-Whitney U/Wilcoxon W	0/210

Features	CTC	Test results	
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
	20	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.411
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	2/212
		Z	-5.356
		Asymp. Sig. (2-tailed)	0
100	10	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
	20	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	2/212
		Z	-5.356
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
150	10	Mann-Whitney U/Wilcoxon W	6/216
		Z	-5.248
		Asymp. Sig. (2-tailed)	0
	20	Mann-Whitney U/Wilcoxon W	1/211
		Z	-5.383
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
	40	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
200	10	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41

Features	CTC	Test results	
		Asymp. Sig. (2-tailed)	0
	20	Mann-Whitney U/Wilcoxon W	4/214
		Z	-5.302
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	0/210
		Z	-5.41
		Asymp. Sig. (2-tailed)	0
250	10	Mann-Whitney U/Wilcoxon W	12/222
		Z	-5.085
		Asymp. Sig. (2-tailed)	0
	20	Mann-Whitney U/Wilcoxon W	22/232
		Z	-4.815
		Asymp. Sig. (2-tailed)	0
	30	Mann-Whitney U/Wilcoxon W	75/285
		Z	-3.381
		Asymp. Sig. (2-tailed)	0.001
(a). Not corrected for ties.			
(b). Grouping Variable: Algorithm (random or evolutionary)			

Table D.2: Experiment #2 Test Statistics

Appendix E
Acronyms

BDD. Binary Decision Diagram.

BeTTy. BEnchmarking and TesTing on the analYsis of feature models.

CNF. Conjunctive Normal Form.

CSP. Constraint Satisfaction Problem.

FaMa. FeAture Model Analyzer.

FM. Feature Model

SAT. Satisfiability Problem.

SPL. Software Product Line.

SPLOT. Software Product Lines Online Tools.

RACER. Renamed ABox and Concept Expression Reasoner.

SXFM. Simple XML Feature Model format.

XML. eXtensible Mark–up Language.

Bibliography

[1] L. Abo, F. Kleinermann, and O. D. Troyer. Applying semantic web technology to feature modeling. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1252–1256, New York, NY, USA, 2009. ACM. DOI: 10.1145/1529282.1529563

[2] D. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987

[3] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009. DOI: 10.1016/j.infsof.2008.12.005

[4] AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html, accessed October 2010

[5] R. T. Alexander, J. Bieman, S. Ghosh, and B. Ji. Mutation of java objects. *International Symposium on Software Reliability Engineering*, 0:341, 2002. DOI: 10.1109/ISSRE.2002.1173285

[6] Alloy Analyzer, http://alloy.mit.edu/, accessed January 2010

[7] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, New York, NY, USA, 2005. ACM. DOI: 10.1145/1062455.1062530

[8] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003

[9] R. Bachmeyer and H. Delugach. A conceptual graph approach to feature modeling. In *Proceedings of the 15th International Conference on Conceptual Structures (ICCS)*, pages 179–191, Berlin, Heidelberg, 2007. Springer-Verlag. DOI: 10.1007/978-3-540-73681-3_14

[10] T. Back, D. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK, 1997

[11] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer–Verlag, 2005. DOI: 10.1007/11554844_3

[12] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December:45–47, 2006. DOI: 10.1145/1183236.1183264

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

[14] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990

[15] D. Benavides. *On the Automated Analyisis of Software Product Lines using Feature Models. A Framework for Developing Automated Tool Support.* PhD thesis, University of Seville, 2007

[16] 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

[17] 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

[18] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *17th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *Lecture Notes in Computer Sciences*, pages 491–503. Springer–Verlag, 2005. DOI: 10.1007/11431855_34

[19] 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

[20] 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, Sitges, Barcelona, Spain, 2006

[21] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010. DOI: 10.1016/j.is.2010.01.001

[22] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006

[23] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP solvers in the automated analyses of feature models. *LNCS*, 4143:389–398, 2006. DOI: 10.1007/11877028

[24] 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

[25] BeTTy Framework. http://www.isa.us.es/betty, accessed November 2010

[26] S. Beydeda. Self-metamorphic-testing components. In *Computer Software and Applications Conference, Annual International*, pages 265–272, September 2006. DOI: 10.1109/COMPSAC.2006.161

[27] BigLever. Biglever software gears. http://www.biglever.com/, accessed June 2010

[28] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999

[29] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996. DOI: 10.1109/12.537122

[30] J. Bosch. From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference (SPLC)*, pages 111–119, Pittsburgh, PA, USA, 2009. Carnegie Mellon University

[31] J. Bradbury, J. Cordy, and J. Dingel. Exman: A generic and customizable framework for experimental mutation analysis. *Mutation Analysis, Workshop on*, 0:4, 2006. DOI: 10.1109/MUTATION.2006.5

[32] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986

[33] W. Chan, S. Cheung, and K. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4 (2):61–81, 2007

[34] W. Chan, T. Chen, and H. Lu. A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *Proceedings of the Fifth International Conference on Quality Software (QSIC)*, pages 241–249, Washington, DC, USA, 2005. IEEE Computer Society. DOI: 10.1109/QSIC.2005.3

[35] W. Chan, S. Cheung, and K. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Proceedings of the Fifth International Conference on Quality Software (QSIC)*, pages 470–476, Washington, DC, USA, 2005. IEEE Computer Society. DOI: 10.1109/QSIC.2005.67

[36] T. Chen, J. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10(1), 2009. DOI: 10.1186/1471-2105-10-24

[37] T. Chen, S. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report HKUST-CS98-01, University of Science and Technology, Hong Kong, 1998

[38] T. Chen, J. Feng, and T. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th International Computer Software and Applications Conference*, pages 327–333, 2002

[39] T. Chen, D. Huang, T. Tse, and Z. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC)*, pages 569–583, 2004

[40] T. Chen, F. Kuo, Y. Liu, and A. Tang. Metamorphic testing and testing with special values. In *Proceedings of the 5th International Conference on Software Engineering, Artificial Intelligence, Networking and Paralell/Distributed Computing*, 2004

[41] T. Chen, T. Tse, and Z. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 191–195. ACM, 2002

[42] T. Chen, T. Tse, and Z. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003

[43] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):90–103, November 2003. DOI: 10.1007/s10009-002-0099-9

[44] Choco solver. http://choco.emn.fr/, accessed October 2010

[45] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, August 2001

[46] S. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971

[47] L. Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Inc., Norwood, MA, USA, 2003

[48] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. ADDISON, May 2000

[49] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Third Software Product Line Conference (SPLC)*, pages 266–282. Springer, LNCS 3154, 2004

[50] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005

[51] 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

[52] M. Dean and G. Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004

[53] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978. DOI: 10.1109/C-M.1978.218136

[54] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991. DOI: 10.1109/32.92910

[55] A. Derezinska and A. Szustek. Tool-supported advanced mutation approach for verification of C# programs. In *Proceedings of the Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, pages 261–268, Washington, DC, USA, 2008. IEEE Computer Society. DOI: 10.1109/DepCoS-RELCOMEX.2008.51

[56] O. Djebbi, C. Salinesi, and D. Diaz. Deriving product line requirements: the red-pl guidance approach. *Asia-Pacific Software Engineering Conference*, 0:494–501, 2007. DOI: 10.1109/ASPEC.2007.63

[57] L. Etxeberria, G. Sagardui, and L. Belategi. Modelling variation in quality attributes. In *First International Workshop on Variability Modelling of Software–intensive Systems (VaMoS)*, 2007

[58] FaMa Tool Suite. http://www.isa.us.es/fama/, accessed October 2010

[59] S. Fan and N. Zhang. Feature model based on description logics. In *Knowledge-Based Intelligent Information and Engineering Systems*, 2006

[60] D. Fernandez-Amoros, R. Heradio, and J. Cerrada. Inferring information from feature diagrams to product line economic models. In *Proceedings of the Sofware Product Line Conference*, 2009

[61] Feature Modeling Plug-in. http://gp.uwaterloo.ca/fmp/, accessed October 2010

[62] P. Frankl, S. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997. DOI: 10.1016/S0164-1212(96)00154-9

[63] J. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. towards automated analysis. In *Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications (ACoTA)*, Antwerp, Belgium, 2010

[64] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *Proceedings of the ACM SIGSOFY First Alloy Workshop*, pages 71–80, Portland, United States, nov 2006

[65] R. Gheyi, T. Massoni, and P. Borba. Algebraic laws for feature models. *Journal of Universal Computer Science*, 14(21):3573–3591, 2008. DOI: 10.3217/jucs-014-21-3573

[66] GNU Prolog, http://www.gprolog.org, accessed October 2010

[67] D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. San Francisco, CA: Morgan Kaufmann, 1991

[68] A. Gotlieb and B. Botella. Automated metamorphic testing. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 34 – 40, 3-6 2003

[69] Graphviz. . http://www.graphviz.org/, accessed October 2010

[70] M. Grindal, J. Offutt, and S. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005. DOI: 10.1002/stvr.319

[71] 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

[72] B. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Proceedings of the 4th International Workshop on Mutation Testing*, 2009

[73] G. Halmans and K. Pohl. Communicating the variability of a software–product family to customers. *Journal on Software and Systems Modeling*, 2(1):15–36, 2003

[74] A. Hemakumar. Finding contradictions in feature models. In *First International Workshop on Analyses of Software Product Lines (ASPL)*, pages 183–190, 2008

[75] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Focus*, 9(4):233–262, 1999

[76] J. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975

[77] IBM. SPSS 17 Statistical Package. http://www.spss.com/, accessed November 2010

[78] 829-2008. IEEE standard for software and system test documentation. Technical report, 2008. DOI: 10.1109/IEEESTD.2008.4578383

[79] JaCoP. http://jacop.osolpro.com/, accessed October 2010

[80] JavaBDD. http://javabdd.sourceforge.net/, accessed October 2010

[81] Jester. http://jester.sourceforge.net/, accessed May 2010

[82] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 99, 2010. DOI: 10.1109/TSE.2010.62

[83] Jumble. http://jumble.sourceforge.net/, accessed May 2010

[84] JUnit. http://www.junit.org/, accessed October 2010

[85] 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, SEI, 1990

[86] K. Kang, J. Lee, and P. Donohoe. Feature–Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, August 2002

[87] G. Kapfhammer. *The Computer Science Handbook*, chapter Software Testing. CRC Press, edition 2nd, June, 2004

[88] A. Karatas, H. Oguztüzün, and A. Dogru. Mapping extended feature models to constraint logic programming over finite domains. In *4th International Software Product Lines Conference*, pages 286–299, 2010

[89] S. Kim, J. Clark, and J. Mcdermid. Assessing test set adequacy for object-oriented programs using class mutation. In *Proceedings of the 3rd Symposium on Software Technology (SoST)*, pages 72–83, Buenos Aires, Argentina, September 1999

[90] S. Kim, J. Clark, and J. Mcdermid. Investigating the effectiveness of object-oriented testing strategies with the mutation method. *Software Testing, Verification and Reliability*, 11:207–225, 2001

[91] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993. DOI: 10.1145/151257.151260

[92] J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992

[93] S. Lau. Domain analysis of e-commerce systems using feature–based model templates. master's thesis. Dept. of ECE, University of Waterloo, Canada, 2006

[94] F. Loesch and E. Ploedereder. Optimization of variability in software product lines. In *Proceedings of the 11th International Software Product Line Conference (SPLC)*, pages 151–162, Washington, DC, USA, 2007. IEEE Computer Society. DOI: 10.1109/SPLINE.2007.31

[95] Y. Ma, M. Harrold, and Y. Kwon. Evaluation of mutation testing for object-oriented programs. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 869–872, New York, NY, USA, 2006. ACM. DOI: 10.1145/1134285.1134437

[96] Y. Ma, Y. Kwon, and S. Kim. Statistical investigation on class mutation operators. *ETRI Journal*, 31:140–150, 2009

[97] Y. Ma, Y. Kwon, and J. Offutt. Inter-class mutation operators for java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)*, page 352, Washington, DC, USA, 2002. IEEE Computer Society

[98] Y. Ma and J. Offutt. Description of class mutation operators for java, http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf, 2005

[99] Y. Ma and J. Offutt. Description of method-level mutation operators for java, http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf, 2005

[100] Y. Ma, J. Offutt, and Y. Kwon. Mujava: An automated class mutation system. *Software Testesting, Verification and Reliability*, 15(2):97–133, 2005. DOI: 10.1002/stvr.v15:2

[101] M. Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second Software Product Line Conference (SPLC)*, LNCS 2379, pages 176–187, San Diego, CA, 2002. Springer

[102] M. Mannion and J. Camara. Theorem proving for product line model verification. In *Software Product-Family Engineering (PFE)*, volume 3014 of *Lecture Notes in Computer Science*, pages 211–224. Springer Berlin / Heidelberg, 2003. DOI: 10.1007/b97155

[103] F. Marić. Formalization and implementation of modern sat solvers. *Journal of Automated Reasoning*, 43(1):81–119, June 2009. DOI: 10.1007/s10817-009-9127-8

[104] P. R. Mateo, M. Usaola, and A. Offutt. Mutation at system and functional levels. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION)*, Paris, France, 6 April 2010

[105] P. McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing Verification and Reliability.*, 14(2):105–156, 2004. DOI: 10.1002/stvr.v14:2

[106] M. Mendonca, T. Bartolomei, and D. Cowan. Decision-making coordination in collaborative product configuration. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC)*, pages 108–113, New York, NY, USA, 2008. ACM. DOI: 10.1145/1363686.1363715

[107] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 761–762, Orlando, Florida, USA, October 2009. ACM. DOI: 10.1145/1639950.1640002

[108] M. Mendonca, D. Cowan, W. Malyk, and T. Oliveira. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *Journal of Software*, 3(2):69–82, 2008

[109] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT–based analysis of feature models is easy. In *Proceedings of the International Sofware Product Line Conference (SPLC)*, 2009

[110] M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan. Efficient compilation techniques for large scale feature models. In *7th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 13–22, 2008. Available at 10.1145/1449913.1449918

[111] D. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*, volume 1. The MIT Press, edition 1, 2005

[112] Metrics, http://metrics.sourceforge.net/, accessed May 2010

[113] Moskitt Feature Modeler. http://www.pros.upv.es/mfm, accessed October 2010

[114] C. Murphy and G. Kaiser. Metamorphic runtime checking of non-testable programs. Technical report cucs-012-09, Dept. of Computer Science, Columbia University, 2009

[115] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 189–200, New York, NY, USA, 2009. ACM. DOI: 10.1145/1572272.1572295

[116] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Conference on Software Testing, Verification, and Validation*, volume 0, pages 436–445, Los Alamitos, CA, USA, 2009. IEEE Computer Society. DOI: 10.1109/ICST.2009.19

[117] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004

[118] S. Nakajima. Non-clausal encoding of feature diagram for automated diagnosis. In *14th International Software Product Lines Conference*, pages 420–424, 2010

[119] S. Nakajima. Semi-automated diagnosis of foda feature diagram. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 2191–2197, New York, NY, USA, 2010. ACM. DOI: 10.1145/1774088.1774550

[120] A. Offutt and W. Craft. Using compiler optimization techniques to detect equivalent mutants. *Journal of Software Testing, Verification, and Reliability*, 4:131–154, 1994

[121] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996. DOI: 10.1145/227607.227610

[122] A. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997. DOI: 10.1002/(SICI)1099-1689(199709)7:3%3C165::AID-STVR143%3E3.0.CO;2-U

[123] J. Offutt, Y. Ma, and Y. Kwon. The class-level mutants of mujava. In *Proceedings of the 2006 international workshop on Automation of Software Test (AST)*, pages 78–84, New York, NY, USA, 2006. ACM. DOI: 10.1145/1138929.1138945

[124] OPL studio, http://www.ilog.com/products/oplstudio/, accessed January 2010

[125] A. Osman, S. Phon-Amnuaisuk, and C. Ho. Knowledge based method to validate feature models. In *First International Workshop on Analyses of Software Product Lines*, pages 217–225, 2008

[126] A. Osman, S. Phon-Amnuaisuk, and C. Ho. Using first order logic to validate feature model. In *Third International Workshop on Variability Modelling in Software-intensive Systems (VaMoS)*, pages 169–172, 2009

[127] 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*. LNCS, 2006

[128] Pellet: the open source OWL reasoner, http://clarkparsia.com/pellet/, accessed October 2010

[129] K. Pohl, G. Böckle, , and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer–Verlag, 2005

[130] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGrap-Hill, edition fifth, 2001

[131] pure::variants. http://www.pure-systems.com/, accessed October 2010

[132] RACER, http://www.racer-systems.com/, accessed January 2010

[133] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987. DOI: 10.1016/0004-3702(87)90062-2

[134] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with uml multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT)*, June 2002

[135] M. Riebisch, D. Streitferft, and I. Pashov. Modeling variability for object-oriented product lines. In *ECOOP 2003 Workshop Reader*. Springer LNCS, 2003

[136] F. Roos-Frantz and S. Segura. Automated analysis of orthogonal variability models. a first step. In S. Thiel and K. Pohl, editors, *In proceedings of the First Workshop on Analyses of Software Product Lines (ASPL)*, pages 243–248, Limerick, Ireland, September 2008

[137] M. Sannella. The skyblue constraint solver and its applications. In *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*, pages 385–406. MIT Press, 1993

[138] Sat4j. http://www.sat4j.org/, accessed May 2010

[139] S. Schach. Testing: principles and practice. *ACM Comput. Surv.*, 28(1):277–279, 1996. DOI: 10.1145/234313.234422

[140] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb 2007. DOI: 10.1016/j.comnet.2006.08.008

[141] D. Schuler and A. Zeller. (un-)covering equivalent mutants. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 45–54, Washington, DC, USA, 2010. IEEE Computer Society. DOI: 10.1109/ICST.2010.30

[142] S. Segura. Automated analysis of feature models using atomic sets. In *First Workshop on Analyses of Software Product Lines (ASPL)*, pages 201–207, Limerick, Ireland, September 2008

[143] S. Segura, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools. a first step. In P. Knauber, A. Metzger, and J. McGregor, editors, *Fifth International Workshop on Software Product Lines Testing (SPLiT)*, pages 36–39, Limerick, Ireland, September 2008

[144] S. Segura, D. Benavides, A. Ruiz-Cortés, and M. Escalona. From requirements to web system design. an automated approach using graph transformations. In A. Estévez, V. Pelechano, and A. Vallecillo, editors, *Actas de Talleres de Ingeniería del Software y Bases de Datos*, volume 1, pages 61–69, Zaragoza. Spain, Semptember 2007

[145] S. Segura, D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Open source tools for software product line development. In *Open Source and Product Lines (OSSPL)*, Kyoto. Japan, September 2007

[146] S. Segura, D. Benavides, A. Ruiz-Cortés, and P. Trinidad. A taxonomy of variability in web service flows. In S. Cohen and R. Krut, editors, *Service Oriented Architectures and Product Lines (SOAPL)*, Kyoto. Japan, September 2007

[147] S. Segura, D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Toward automated refactoring of feature models using graph transformations. In E. Pimentel, editor, *VII Jornadas sobre Programación y Lenguajes (PROLE)*, pages 275–284, Zaragoza. Spain, September 2007

[148] S. Segura, D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated merging of feature models using graph transformations. *Post-proceedings of the Second Summer School on Generative and Transformational Techniques in Software Engineering*, LNCS, 5235: 489–505, 2008

[149] S. Segura, D. Benavides, and A. Ruiz-Cortés. FaMa Test Suite v1.2. Technical report ISA-10-TR-01, ISA Research Group, 2010. Available at http://www.isa.us.es/

[150] S. Segura, D. Benavides, and A. Ruiz-Cortés. Functional testing of feature model analysis tools: A test suite. *IET Software*, 2010 (in press)

[151] S. Segura, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *International Conference on Software Testing, Verification and Validation*, pages 35–44, Paris, France, 2010. IEEE press. DOI: 10.1109/ICST.2010.20

[152] S. Segura, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology Special Issue on Mutation Testing*, 2010 (in press)

[153] S. Segura, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53:245–258, 2011. DOI: 10.1016/j.infsof.2010.11.002

[154] S. Segura and A. Ruiz-Cortés. Benchmarking on the automated analyses of feature models: A preliminary roadmap. In *Third International Workshop on Variability Modelling of Software-intensive Systems*, pages 137–143, Seville, Spain, 2009

[155] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the linux kernel. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Linz, Austria, January 2010

[156] B. Smith and L. Williams. An empirical evaluation of the mujava mutation operators. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 0:193–202, 2007. DOI: 10.1109/TAIC.PART.2007.12

[157] B. Smith and L. Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, 2009. DOI: 10.1007/s10664-008-9083-7

[158] SMV system , http://www.cs.cmu.edu/~modelcheck, accessed January 2010

[159] S.P.L.O.T.: Software Product Lines Online Tools. http://www.splot-research.org/, accessed October 2010

[160] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber. Introducing pla at bosch gasoline systems: Experiences and practices. In *International Sofware Product Line Conference (SPLC)*, pages 34–50, 2004

[161] 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

[162] V. Sugumaran, S. Park, and K. Kang. Software product line engineering. *Commun. ACM*, 49(12):28–32, 2006. DOI: 10.1145/1183236.1183260

[163] J. Sun, H. Zhang, Y. Li, and H. Wang. Formal semantics and verification for feature modeling. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005

[164] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *International Conference on Software Engineering*, pages 254–264, 2009. DOI: 10.1109/ICSE.2009.5070526

[165] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81 (6):883–896, 2008. DOI: 10.1016/j.jss.2007.10.030

[166] 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 Workshop on Decision Support in Software Engineering*, volume 120. CEUR Workshop Proceedings (CEUR-WS.org), 2004

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

[168] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A.Jimenez. Fama framework. In *12th Software Product Lines Conference (SPLC)*, page 359, 2008. DOI: 10.1109/SPLC.2008.50

[169] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and M. Toro. Explanations for agile feature models. In *Procceedings of the 1st International Workshop on Agile Product Line Engineering (APLE)*, Baltimore, Maryland, USA, 2006

[170] P. Trinidad, D. Benavides, S. Segura, and A. Ruiz-Cortés. Fama: hacia el análisis automático de modelos de características. In *Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos (demostración de herramientas).*, 2007

[171] P. Trinidad, A. Ruiz-Cortés, D. Benavides, and S. Segura. Three-dimensional feature diagrams visualization. In *2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008)*, Limerick, Ireland, 2008

[172] P. Trinidad and A. R. Cortés. Abductive reasoning and automated analysis of feature models: How are they connected? In *Third International Workshop on Variability Modelling of Software-Intensive Systems. Proceedings*, pages 145–153, 2009

[173] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995

[174] M. Tseng and J. Jiao. *Handbook of Industrial Engineering: Technology and Operations Management*, chapter Mass Customization, page 685. Wiley, 2001

[175] I. Valiela. *Doing science: design, analysis, and communication of scientific research*. Oxford University Press New York, 2001

[176] P. van den Broek and I. Galvao. Analysis of feature models using generalised feature trees. In *Third International Workshop on Variability Modelling of Software-intensive Systems*, number 29. In ICB-Research Report, pages 29–35, Essen, Germany, January 2009. Universität Duisburg-Essen

[177] T. van der Storm. Variability and component composition. In *8th International Conference on Software Reuse: Methods, Techniques and Tools (ICSR)*, volume 3107 of *Lecutre Notes in Computer Sciences*, pages 157–166. Springer, July 2004

[178] T. van der Storm. Generic feature-based software composition. In *Software Composition*, volume 4829 of *LNCS*, pages 66–80. Springer, 2007

[179] A. van Deursen and P. Klint. Domain–specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002

[180] 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)*, LNCS 3014, Siena, Italy, 2003. Springer Verlag

[181] T. von der Massen and H. Litcher. Determining the variation degree of feature models. In *Software Product Lines Conference, LNCS 3714*, pages 82–88, 2005

[182] S. Voß. Meta-heuristics: The state of the art. In *Proceedings of the Workshop on Local Search for Planning and Scheduling-Revised Papers in ECAI*, pages 1–23. Springer-Verlag, London, UK, 2001

[183] B. Wang, Z. Hu, Y. Xiong, H. Zhao, W. Zhang, and H. Mei. Tolerating inconsistency in feature models. In *3rd Workshop on Living With Inconsistency in Software Development, held with 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010

[184] B. Wang, Y. Xiong, Z. Hu, H. Zhao, W. Zhang, and H. Mei. A dynamic-priority based approach to fixing inconsistent feature models. In D. Petriu, N. Rouquette, and O. Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 181–195. Springer Berlin / Heidelberg, 2010. DOI: 10.1007/978-3-642-16145-2_13

[185] 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)*, November 2005

[186] H. Wang, Y. Li, J. un, H. Zhang, and J. Pan. Verifying Feature Models using OWL. *Journal of Web Semantics*, 5:117–129, June 2007. DOI: 10.1016/j.websem.2006.11.006

[187] J. Wegener, K. Grimm, M. Grochtmann, and H. Sthamer. Systematic testing of real-time systems. In *Proceedings of the Fourth International Conference on Software Testing and Review (EuroSTAR)*, 1996

[188] J. Wegener, H. Sthamer, B. Jones, and D. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Control*, 6(2):127–135, 1997. DOI: 10.1023/A:1018551716639

[189] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982

[190] J. White, D. Benavides, D. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83 (7):1094–1107, 2010. DOI: 10.1016/j.jss.2010.02.017

[191] J. White, B. Doughtery, and D. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009. DOI: 10.1016/j.jss.2009.02.011

[192] J. White, B. Doughtery, D. Schmidt, and D. Benavides. Automated reasoning for multi-step software product-line configuration problems. In *Proceedings of the Sofware Product Line Conference*, pages 11–20, 2009

[193] J. White and D. Schmidt. Filtered cartesian flattening: An approximation technique for optimally selecting features while adhering to resource constraints. In *First International Workshop on Analyses of Software Product Lines (ASPL)*, pages 209–216, 2008

[194] J. White, D. Schmidt, D. B. P. Trinidad, and Ruiz-Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the 12th Sofware Product Line Conference (SPLC)*, Limerick, Ireland, September 2008

[195] X. Xie, W. Wong, T. Chen, and B. Xu. Spectrum-Based Fault Localization Without Test Oracles. Technical report, Technical Report, UTDCS-7-10, Department of Computer Science, University of Texas at Dallas, 2010

[196] H. Yan, W. Zhang, H. Zhao, and H. Mei. An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. In *ICSR*, pages 65–75, 2009. DOI: 10.1007/978-3-642-04211-9_7

[197] 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. DOI: 10.1007/s00766-006-0033-x

[198] 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. DOI: 10.1007/b102837

[199] W. Zhang, H. Yan, H. Zhao, and Z. Jin. A bdd–based approach to verifying clone-enabled feature models' constraints and customization. In *10th International Conference on Software Reuse (ICSR)*, LNCS, pages 186–199. Springer, 2008. DOI: 10.1007/978-3-540-68073-4_18

[200] Z. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology*, pages 346–351, 2004

[201] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. DOI: 10.1145/267580.267590

This document was typeset on 2011/1/31 using RC–BOOK α2.11 for $\LaTeX_{2\epsilon}$. Should you want to use this document class, please send mail to contact@tdg-seville.info.