Configuration of hyper-graph based feature diagrams

Guillermo Rodríguez Cano
Abstract

Configuration of hyper-graph based feature diagrams

Guillermo Rodríguez Cano

Software product line engineering has gained an exceptional attention and interest from scientific community in recent years as a consequence of reuse in mass software production.

However, management of the common and variable characteristics and functionalities among a collection of software systems, which belong to the same application domain, is still a work in progress because it is neither a trivial activity nor can it be resolved applying conventional software engineering methodologies.

One of the suggested solutions, and generally accepted, for elicitation, representation and management of variability among software systems given a domain is feature modelling. Feature diagram, the fundamental graphical representation of feature modelling, proposed two decades ago in the study “Feature-Oriented Domain Analysis” published by the Software Engineering Institute, is a powerful representation tool of product lines’ variability, but at the same time a systematic element for product configuration.

First proposal of feature diagram defined the underlying data structure by means of trees, while later proposals used (directed acyclic) graphs or defined a new semantic meaning for trees. Nevertheless, there is yet to be found a consensus for a definition of feature diagrams.

Regardless of the definition, any representation ought to be accompanied by a methodology to not only validate the model but also assist with the configuration of a product, because it is part of any Computer Aided Software Engineering tool.

This work focuses on a mathematical representation of feature diagrams, (acyclic forward) hyper-graphs, and takes advantage of their properties and existing traversal algorithms to propose simple and robust procedures to aid with the process of configuration.

The first part reviews previous definitions of feature diagram and describes the formalism proposed within GIRO research group as well as two-phases configuration algorithms to generate partial configurations and to complete them. Second part empirically evaluates the validity of these algorithms for the hyper-graph formalism and assesses the performance of a representative set of selected test feature models.

The outcome will be a detailed study of feature modelling variability mechanisms in software product lines along with a mathematical depiction of feature diagrams, which overcomes the problem of denoting the semantics of constraints when using graphs and trees representations, suitable and feasible validation and configuration phases algorithms and an empirical assessment of the proposed algorithms.
This thesis is dedicated to each single human being, whose sole act of reading these lines contributes to the progress of society, which I have profoundly benefited from.
Acknowledgements

Thanks Miguel and José Manuel, for all those hours you have dedicated to help me understand the quirks of hyper-graphs. Thanks for your explanations, clarifications and support. And particularly, thanks for your patience during all these past months.

Thanks to my reviewer, Lars-Henrik, for your insightful and constructive comments while assessing this thesis.

To my mom and dad, who have supported me during this new chapter of my life. Once again, no matter how many times I thank you it will always be insufficient.

To my brother Mario, thanks for bringing a little piece of encouragement during the visit at the beginning of this challenging adventure.

Thanks João, Thành and Vasilij for sharing this path of life and letting me walk with you. It has been a pleasure.

I should also show my gratitude to Fei, Erika, Seif, Cecilia, Juan and those who have made this couple of years heartening and inspiring.

Last, but not least, thanks to the “Cuadrilla”, you have borne with me these months and I know it has been a mission requiring some patience.
# Contents

1 Introduction

1.1 Motivation and goals .................................................. 4

1.2 Summary of chapters .................................................. 6

2 Background and methodology ........................................... 7

2.1 Model-driven product line engineering ............................. 7

2.1.1 Motivations ......................................................... 8

2.1.2 Definition ........................................................ 9

2.1.3 Architecture ...................................................... 11

2.1.4 Development process ............................................ 12

2.1.5 Benefits ........................................................ 13

2.1.6 Quality ........................................................ 13

2.2 Variability in software product lines .............................. 15

2.2.1 Origins of variability management ............................ 15

2.2.2 Feature-based modelling ........................................ 16

2.3 Directed hyper-graphs modelling paradigm ...................... 24

2.3.1 Graph definition ............................................... 24

2.3.2 Undirected hyper-graphs ...................................... 25

2.3.3 Directed hyper-graphs ......................................... 26

2.3.4 Applications .................................................... 29

2.4 Work methodology .................................................... 31

2.4.1 Delimitations .................................................... 31

3 Feature diagrams and hyper-graphs ................................ 33

3.1 Tree and graph modelling of feature diagrams ................. 33

3.2 Directed hyper-graph modelling of feature diagrams ....... 36

4 Verification of feature diagrams ...................................... 41

4.1 Graph traversal ......................................................... 41

4.1.1 Applications .................................................... 41

4.2 Hyper-graph traversal ................................................ 43

4.2.1 Visit ............................................................. 43

4.2.2 Backward visit .................................................. 44

4.3 Verification and configuration ...................................... 46

4.3.1 Configuration process ......................................... 47

4.4 Experimental evaluation ............................................. 52

4.4.1 Technical details ............................................... 52

4.4.2 Results ........................................................ 56

5 Conclusions and future work .......................................... 67

5.1 Future work .......................................................... 68
# List of Figures

1.1 Map of variability management approaches ........................................... 2
2.1 Diagram of commonalities and variabilities among three products ............... 8
2.2 Software product line abstract scheme ................................................ 11
2.3 Process flow of the phases in an (object-oriented) software product line .......... 12
2.4 Feature diagram in FODA notation ..................................................... 16
2.5 Feature diagram of a search engine product line ..................................... 18
2.6 Equivalent feature diagram from figure 2.5 using extended Czarnecki-Eiselecker notation ............................................................. 21
2.7 Staged configuration of feature diagram ................................................ 23
2.8 Euler’s Königsberg graph problem ....................................................... 24
2.9 Hyper-graph (and equivalent graph) of a protein-protein network interaction .... 25
2.10 Directed hyper-graph ................................................................. 26
2.11 Types of directed hyper-arcs .......................................................... 27
2.12 Transformation of a hyper-arc into a BF-hyper-arc .................................... 28
2.13 Hyper-path example ................................................................. 28
3.1 Example illustrating Riebisch et al. feature diagram graph representation .......... 35
3.2 Telecommunications system feature diagram and (hidden) hyper-graph transformation ........................................................................... 37
3.3 Telecommunications system f-graph transformation .................................... 38
4.1 Feature diagram of a graph product line ............................................... 53
4.2 Debugging feature diagram sample ..................................................... 53
4.3 Class diagram of feature diagram and hyper-graph representations .............. 55
B.1 Software Product Line’s proactive approach .......................................... 79
B.2 Software Product Line’s reactive approach .......................................... 79
B.3 Software Product Line’s extractive approach .......................................... 80
B.4 Software Product Line’s refactorive approach ........................................ 80
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Time measurements for core configuration on all test feature models</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Time measurements for staged configuration on small-sized test feature models</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Time measurements for staged configuration on medium-sized test feature models</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Time measurements for staged configuration on big-sized test feature models</td>
<td>62</td>
</tr>
<tr>
<td>4.5</td>
<td>Time measurements on benchmark feature models</td>
<td>63</td>
</tr>
<tr>
<td>4.6</td>
<td>Time measurements on customised test feature model for multiple parent features</td>
<td>64</td>
</tr>
<tr>
<td>A.1</td>
<td>FODA derived variability management approaches</td>
<td>77</td>
</tr>
<tr>
<td>C.1</td>
<td>Conversion of two clauses CNF to feature diagram, graphical and textual constraint</td>
<td>82</td>
</tr>
<tr>
<td>C.2</td>
<td>Conversion of three clauses CNF to feature diagram and textual constraint</td>
<td>83</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1</td>
<td>Hyper-graph visit procedure</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Hyper-graph b-visit procedure</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Feature diagram (partial) configuration procedure</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Feature diagram completing (partial) configuration procedure</td>
<td>51</td>
</tr>
<tr>
<td>4.5</td>
<td>XML model of graph product line feature diagram</td>
<td>54</td>
</tr>
<tr>
<td>D.1</td>
<td>Breadth-first search procedure</td>
<td>85</td>
</tr>
<tr>
<td>D.2</td>
<td>Depth-first search procedure</td>
<td>86</td>
</tr>
<tr>
<td>D.3</td>
<td>Depth-first search procedure for directed acyclic graphs</td>
<td>86</td>
</tr>
<tr>
<td>D.4</td>
<td>Dijkstra’s procedure for directed acyclic graphs</td>
<td>87</td>
</tr>
<tr>
<td>D.5</td>
<td>Single-source shortest path procedure for directed acyclic graphs</td>
<td>87</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“The past is but the beginning of a beginning, and all that is or has been is but the twilight of the dawn”

Herbert G. Wells

One of the goals, if not the most important one, in modern software engineering is reuse, particularly in industrial software development processes in view of the fact that it contributes towards saving resources (human and material), time and, particularly, budget.

But the advantages are not exclusively economical; at the same time, reuse and development policies towards reuse improve the overall performance of software products and strengthens their quality. Object-oriented development has to some extent an aim for reuse, because many object-oriented programming languages provide with structures that can be extended incrementally, so that not only components are reused but also their functionality is broaden and expanded.

However, most of the progress in reuse, which object orientation has generated, is reaching its own limits. For instance, classes, as unitary elements of reuse, are too small; libraries, as independent software components with their own interfaces have a better potential of reuse, but at a cost of increased complexity.

That being so, an alternative technique, procedure or methodology would be desirable in order to change the trend and increase productivity without proportionally magnifying the costs.

Software product line development is a software engineering paradigm which takes advantage of reuse regardless of how complex the domain scope is [67]. Product lines can create a collection of systems where all of them share some common elements, this is called commonality; but each system has at least one component or characteristic which makes it different from the others, this is denominated variability.

Product lines can be perceived as a two life-cycle processes’ flow[2] where the first is called domain engineering and the second is denominated application engineering. The former deals with the development of the product line itself; that is, it defines the domain scope and makes sure that all required assets are available. And the latter process flow deals mostly with the production of each product.

Nevertheless, the crucial phase in both flows is the management of product line variability. In the former process flow, requirements elicitation will play an important role for variability points recognition and variability design; while in the latter process flow, designers, customers and end-users will have the opportunity to choose the features and characteristics among the ones that the former phase identified and mapped, they want to have for their products.

Scientific literature distinguishes two varieties of variability in software product line engineering depending on the building block under consideration:

1Some authors use the term family system instead
2See figure 2.3 for an illustration depicting the interactions between these two processes
CHAPTER 1. INTRODUCTION

- **Software variability** [10, 82]
  
  Svahnberg et al. define it as the “ability of a software system or artefact to be changed, customised, extended or configured for use in a particular context”.

- **Product Line variability** [69, 70]
  
  Pohl et al. define it as the “process of documenting the variability among the products which are part of a product line”.

These two categories have a common property, management of variability. Software variability deals with variability at a small scale, “product-scale”, whereas product line variability deals with variability at a big scale, “domain-scale”.

Both types of variability require a correct management for a successful deployment of the software product line and good quality on the products. But it is the product line variability, the class that is key for an effective outcome. This variability is usually managed by means of models, which in software product line engineering is achieved because of feature modelling.

Feature modelling, an amalgamation in one model of various models (structural, behavioural,...), is one of the proposed solutions for elicitation, representation and management of variability and commonality among the collection of software systems known as software product lines, as well as a decision map for the configuration of particular applications (towards their production).

But there is not a common definition for the primary elements of feature modelling; specifically, features and feature diagrams. In 1990, Kang et al. published their Feature-Oriented Domain Analysis (FODA) [40], which was the first proposal that came up with a procedure to handle variability in a graphical way, depicting the common and variable functionalities as a tree of nodes, where nodes symbolised those functionalities (features) and the relationships between each node (edges) emphasised the hierarchical semantic meaning of the decomposition of a feature into subfeatures.

After Kang et al. proposal, many other propositions were presented to the scientific community. Figure 1.1 illustrates some of these proposals, derived from FODA (see appendix A for a table describing each acronym as well as the author(s) and the year it was presented).

Nevertheless most of the proposals depicted in figure 1.1 and others [14, 76], would resolve FODA’s flaws, such as the lack of the OR relationship, or propose alternative underlying structures, such as directed acyclic graphs.
(DAGs) instead of trees, like in FODA.

For instance, graphs are not able to properly describe the multiplicity as a property of the feature; for this reason, Riebisch et al. proposed that arcs should carry the multiplicity instead of the nodes [72]. On the other hand, Schobbens et al. examined different alternative feature diagrams to end up suggesting a non-redundant variant of a feature model (VFD) [76].

Given these distinct ideas, Laguna and Marqués propose in [53] using the mathematical underlying structure of acyclic f-hyper-graphs (a class of directed hyper-graphs) instead of trees or graphs in order to represent a feature diagram. The different decompositions and features can be symbolised with two elements: generic decomposition relationships (hyper-arcs), where each hyper-arc is labelled with a cardinality, and features (nodes).

Subsequently, a feature diagram is originated in a subset of directed hyper-graphs (a “generalisation” of graphs), that is, forward (acyclic) hyper-graphs. This characterisation is suitable for this representation because DAGs or trees cannot symbolise many-to-one relationships, for which hyper-graphs have proved themselves to be more than enough [30], but another original contribution made by Laguna and Marqués with their approach is that the feature model (which is composed of feature diagrams, extra constraints and documentation) can be modelled in one structure as opposed to other prior proposals from scientific literature; and this is what minimises the number of algorithms required as well as the complexity of intermediate structures as everything can be allocated in one model and managed in a simplified manner.

However, any representation (hyper-graphs, DAGs, trees,...) of a feature diagram is not exempt from inconsistencies either when modelling a software product line, or configuring a specific software product line instance, a product. For this reason, once a particular configuration of a software product line has been achieved by one of the stakeholders involved (usually a software engineer), it must be verified for incompatibilities according to the extra constraints that the model portrays; and, in some cases where this selection is valid but it does not define a unique product because it is not complete, it ought to be completed either automatically or by guiding the engineer with the completion.

It is this verification, and especially a formalisation of the technique[3] or process, what the approach proposed by Laguna and Marqués aims at as opposed to the traditional manner of manually verifying single programs vastly researched.

In other words, proving the correctness of the main modeling technique used to represent software product lines, that is feature models, based on hyper-graphs and using the known properties and algorithms that this mathematical model yields to verify not only the feature model itself but also a (user) chosen configuration, but also assist with a plausible completion or correction, in case this choice is unfinished or mistaken.

This verification process is not a trivial task in software product lines because scalability is one of the main contributing factors towards easing a practical and feasible technique as other authors have highlighted in previous work [43, 81] or case studies [4, 13], or formal methods community in related research [36].

Therefore, an automation of this process, which in the case of this thesis is based on graph theory, in particular hyper-graphs, is desirable, and it becomes necessary if a competitive and successful method for industrial software development of software product lines is to be provided.

---

3Formalisation technique understood as a model check provided a finite set of states and another set of logical properties that will be used to check whether they apply for the model in an automated manner as suggested in [33].
1.1 Motivation and goals

The work conducted in this thesis arose from the topic that researchers, Miguel A. Laguna and José M. Marqués Corral, affiliated to “Grupo GIRO” at the Department of Computer Science, University of Valladolid, have been conducting over the past few years in connection with the prominent research field of Software Product Line Engineering and Model-Driven Engineering.

Miguel A. Laguna and José M. Marqués Corral have focused their research efforts on the formalisation of a novel representation for the predominantly decision model used to depict the variability of software product lines, feature models (composed of feature diagrams).

Laguna and Marqués propose in [53] the application of forward hyper-graphs (a class of acyclic directed hyper-graphs [30]) instead of trees or graphs in order to represent feature diagrams.

This project is aimed at finding (and adapting) suitable algorithms that enable an automatic verification of a configuration created from a feature diagram modelled with the underlying data structure of a forward hyper-graph in reasonable time.

Since any particular configuration based on a feature diagram is a subset of the feature diagram, these algorithms should be able to validate any configuration as well.

Not only finding these algorithms is important but also assessing their characteristics within the context of feature modelling. Therefore, realising these algorithms in a general-purpose programming language for experimentation (on selected feature diagram samples) and an analysis of these results will also be part of this project in order to corroborate or refute the initial hypotheses.

Given the previous brief summary of the project, the following objectives of interest emerge:

- **Review of software product line development, and variability management**
  Considering that product line engineering is an area of advance, one important task will be searching, reading, analysing and reviewing scientific literature about software product lines with a special focus on variability management because it is the key question to answer for a successful deployment of a product line.

- **Review of hyper-graph theory and its application to feature models**
  A good understanding of the mathematical theory behind the abstraction of graphs, that is, hyper-graphs, is very important to succeed in the following goal. Therefore, another crucial task will be searching, reviewing, studying and understanding basic hyper-graph theory and its appropriate application to feature modelling as the underlying structure.
  For this objective, it is suggested to start with Gallo et al. proposal [30].

- **Review of graph and hyper-graph traversal algorithms**
  Once the mathematical background has been understood, another proposed task is searching and reviewing scientific literature for graphs and hyper-graph traversal algorithms, to have a notion of some classic algorithms as well as those relevant for the traversal of a feature model represented as a hyper-graph.
  For this objective, the previous suggestion remains as well.

- **Development, experimentation and analysis of verification algorithms for feature models**
  After all theoretical knowledge about variability management with feature models, characterised as forward hyper-graphs, in model-driven software product lines, the last tasks will be the design and implementation of traversal algorithms for verification of a feature model and a (user) chosen configuration.

---

4 Grupo GIRO is the acronym of “Grupo de Investigación en Reutilización y Orientación a Objeto” (Research Group in Reuse and Object Orientation)
1.1. Motivation and Goals

Besides the tasks of implementing, it is required experimentation on selected feature model samples, and analysis of the results; and if necessary, changes on the implementation.

For this objective, Laguna and Marqués proposal [53] shall be used.
1.2 Summary of chapters

A brief synopsis of each chapter, which this thesis is composed of, is listed below in order for the reader to have an overall notion of how this work has been organised in relation to the goals outlined in the previous section.

- **Chapter 2: Background and methodology**
  This chapter introduces the theoretical knowledge about model-driven software product line development, with a focus on the motivations of this emerging methodology, the development process alternatives and the benefits that a software development company would profit from.
  Another topic introduced is variability management in model-driven software product lines based on feature modelling, a methodology that was born in 1990 when Kang et al. published their Feature-Oriented Domain Analysis (FODA). The chapter is concluded with, hopefully not too brief for the reader, a condensed introduction to the mathematical structure of hyper-graphs.

- **Chapter 3: Feature diagrams and hyper-graphs**
  In this chapter thoroughly reviews previous proposals of feature diagram modelling using classical data structures such as trees or directed acyclic graphs and ends with an extensive description of the applied theory for hyper-graph feature diagram modelling based on the proposal by Laguna and Marqués in [53].

- **Chapter 4: Verification of feature diagrams**
  An experimental evaluation of the algorithms proposed in chapter 3 using “real” feature models as well as benchmark and customised models is presented in this chapter along with an analysis of the results.
  A section with details about the experimentation platform and implementation specifics is also included.

- **Chapter 5: Conclusions and future work**
  The final chapter gives an outline of the achieved conclusions once the work has been completed. Conclusions that, beyond being positive or negative, are intended to suggest future work, as well as enhancements or proposed continuations of the work done.

- **Appendix A: Variability management approaches**
  The first appendix chronologically summarises in a table some variability management proposals derived from Feature Oriented Domain Analysis study [40] over the past 20 years, reviewed by Chen et al. in [14].

- **Appendix B: Software product line development processes**
  This appendix illustrates a categorisation (proactive, reactive, extractive and refactorive) that researcher Krueger proposed in [49] for software product line development in compliance with the approach, taken as a model by the development team to engineer the platform of a product line.

- **Appendix C: Conjunctive normal forms feature modelling**
  This other appendix summarises the transformations employed to convert two and three clauses conjunctive normal forms (CNF), used to describe the constraints of some feature models of the sample feature model repository (SPLOT) [60], to the equivalent feature diagram notation.

- **Appendix D: Classical graph traversal algorithms**
  The last appendix illustrates the pseudocode of some classical algorithms for traversing graphs, both directed and undirected, such as Breadth-First Search (BFS), Depth-First Search (DFS) and its variant for directed acyclic graphs (DAG-DFS) and, Dijkstra’s well known algorithm, which is an implementation of Single-Source Shortest Path (SSSP); as well as a SSSP variant for directed acyclic graphs (DAG-SSSP).

---

It is expected from the reader a basic knowledge in data structures such as lists, queues, trees and graphs. It is recommended, but not required, some elemental algebra and graph theory knowledge to ease the process of understanding the mathematical terminology and theory explained in this thesis.
2.1 Model-driven product line engineering

The majority of software development processes are intended to be used by all stakeholders involved in the process of a conventional software development company for the manufacture of individualised (software) “products”, including the techniques that some of these processes may describe as part of their procedure, and the tools used for the development itself.

However, the construction of a software product which shares some functionalities with a previous development (obviously in the same development company), that is, an individualised product having some characteristics in common with a prior project (also individualised) for another (even for the same) customer, may benefit from these commonalities with preceding developments because individualisation of software tends to be expensive (customisation is expensive, and management of each varying characteristic towards a later reuse may require too much effort and be very time consuming) and the usage of standard products usually lacks essential features required by customers.

Consequently, there is a need for a platform in order to solve the problem of mass software development production where the cost of customisation for individual customers results in a small percentage of the total cost of the product while, at the same time, the management of this customisation is feasible in terms of time and effort.

Mass production has proven itself over the course of history to be a process in which hundreds of units of the same product are manufactured or produced, i.e. tire manufacturing, industrial bread bakery. However, mass production was not enough because not only the production of many units was important but also the possibility of customisation of these units, usually to a certain degree.

The era of Industrial Revolution gave rise to mass production where customisation of each produced unit was feasible, i.e. car assembly lines. Analogously, software product lines have mimicked the development process of assembly lines because customisation of software products is a requisite for every development company (as it is producing cars with different colours, number of doors,...), but also automation of this individualisation of characteristics in a platform (as it is for any car manufacturer because it is not feasible to have one assembly line per car model configuration).

That is to say, the main goal of software product lines is the assembly of customised software (reusing the compo-
nents of each software product) in a mechanised and mass production manner.

Mimicking the behaviour of an assembly line does not entirely solve the problem of mass production and variability management. Software engineering has tried to solve this problem before, though it never considered predictability when reusing (components) as an alternative to serviceability. Previous attempts focused on developing a fixed set of functionalities (possible customisations), which were made available as a library, but it could not be demonstrated that all functionality provided was in fact being used. Software product lines opted for a practical approach in which it could be proved that each implemented feature was required by at least one product of the available set of products the product line could produce.

It is important to identify all commonalities among the different products, but the management of variability is vital to reduce the overall cost and development time as figure 2.1 illustrates.

![Diagram of commonalities and variabilities among three products](image)

**Figure 2.1:** Diagram of commonalities and variabilities among three products

### 2.1.1 Motivations

Despite the fact that the first and foremost goal of software product lines is manufacturing individualised products where the cost of this customisation is suitable compared to the entire cost of the software product, there are many other reasons to accomplish the required changes within an organisation to opt for product line engineering.

Some of these arguments are briefly discussed below:

- **Productivity** [18, 84]

  Management of the variability is the most important reason for deploying a platform for product line engineering. Usually, a company has a set of products that are alike but they have been developed in an individual manner, but new products are required which could benefit from these previous developments, and a platform where all common and variable features among these would be very helpful.

  Being able to manage the variability and commonality is key to better productivity but also marketing time, maintenance, complexity or product costs.

---

3Given the amount of texts about product line engineering and mass customisation engineering, only the most relevant motives have been considered, but the reader is more than welcome to further check the given references.
2.1. MODEL-DRIVEN PRODUCT LINE ENGINEERING

• **Quality** [18, 69, 84]

The more products built using a software product line, the more reliable the platform will be because there are more quality tests to carry out which will pinpoint possible errors or problems not only in the characteristics exclusive to the product under testing but also in the platform.

That is, the platform feeds back from the products, thereby increasing the confidence on it because the number of defects decreases over its lifetime (which benefits following products). In the end, customers feel more confident because they get better quality for less money.

• **Marketing** [18, 84]

It is quite obvious that development time of a software product line is much longer than traditional development of a product, because not only the platform and its architecture of a product line must be designed but also the initial components. Once this stage is past, the marketing time of a product is notably shorter.

• **Maintenance** [84]

Any defect discovered during any phase of the development, during deployment too, will be feedback into the product line, so that maintenance work of the platform is reduced.

A good architecture of the platform is key for its success because learning curve for newcomers to the development team will be lessen. But that is why it is so important to develop proper tests when any change is introduced (whether coming from a defect or a new feature).

Designing the platform for its evolution should help reducing the costs of future extensions. Following James O. Coplien advice is quite handy: “same design techniques that lead to good reuse also lead to extensibility and maintainability over time”.

• **Complexity** [69, 84]

The difficulty of the system to be built tends to increase in relation to the number of stakeholders involved because each one of them has a need to be addressed, hence functionality usually grows exponentially. At the same time, the more complex the system is, the wider the scope of customers is.

Given such variety of functionalities and characteristics, and users; the variability of the system might be too complicated for any developer who wants to have an overall idea of the system. Thus, some measures are to be taken in order to ease this job of understanding towards reusability policies and development efficiency.

• **Cost** [69, 84]

One of the most important reasons for product line engineering is the minimisation of costs. While developing components for the platform has a cost, reusing these in different produced systems results in a reduction of the aggregated cost of the platform.

Diverse studies [17, 89] also show that a product line platform is profitable after\(^4\) the third (sometimes the fourth) product.

Companies also have the possibility to sell a product that takes advantage of reuse while they can increase costs for new features not yet available in the product line.

2.1.2 Definition

A software product line, as defined by Clements and Northrop in [17], is “a family of systems which shares a common set of core technical assets, with preplanned extensions and variations to address the needs of specific customers or market segments”.

Regardless of the industrial approach of the definition, a software product line is of extreme importance for software development industry in general and its customers in particular because some aspects such as market analysis,

\(^4\)Considering the development time of new features shorter than a single product

\(^5\)This will naturally be dependent on organisational aspects of the development company as well as the market which the product line aims for.
development for the organisation as well as training (in product line engineering), have become as important as other aspects much more traditional in software engineering development, i.e. architecture.

The development process in a software product line is organised so that reuse is a key factor since the common aspects the product line will include in each product are collected into components; and the engineering process is also structured in such a way that variability among the product line can be easily accommodated and managed with suitable methodologies, techniques and practices.

An abstract overview [16, 17, 48] of a software product line is illustrated in the left-hand side of 2.2. The major sections of interest in a software product line are as follows:

- **Input**
  Set of assets which will be used to create products with the software product line, such as: software components (libraries), requirements, diagrams, techniques, methodologies, tests,...
  Each asset has a role, some of them will be used during the early stages of the product line development (for instance, methodologies and techniques), some others will be used during the entire lifecycle of the software product line (for instance, software components such as graphical libraries) though it is likely that their utilisation is done according to some protocols or guidelines.
  Notwithstanding that there is an important constituent of the input, the model. There is no software product line without a model that depicts the variability in a systematic and manageable manner. This model controls how the software components can be assembled together to materialise the desired product.

- **Configuration**
  Seeing that a software product line constructs a set of products within the bounds of a specific domain, there must be a supported decision model. This decision model describes the optional and mandatory software components to be used.
  Any product generated by the product line has a unique decision model, which is derived from the abstract model depicting the software product line. The resulting model that manages the manufacture of a product has no uncertainty, in other words, all decisions about its variability points have been taken.

- **Production**
  The software product line platform is the place where the process of mass-product manufacture happens. Some of the techniques and methodologies which were queued in the input section are put into use in this important stage to assemble the software components following the decisions model, that is driving which components are to be included and which ones are not going to be part.

- **Output**
  Finally, once the production platform is finished with the assembly of some of the input components, the result will be the “product”.
  However, the purpose of a software product line is not merely the generation of end products. A software product line can be designed so that the results are more software components, which may be part of another software product line (or even the same product line).

Right-hand side in figure 2.2 exemplifies the three main activities that Northrop considers essential to the overall success of a software product line. It is easy to understand that a product line feedbacks itself at all times, and an appropriate management activity, which drives the product line engineering process, is crucial for a long lifecycle.

Even though Northrop specifies two separate development activities, one for the assets (input software components) and another one for the products development; these two may be merged together into one development

---

6Section 2.2 describes the model often used in model-driven software product line engineering
activity, or the asset development activity can be another product line feeding the input of the one producing the end-products; but the goal, in the end, is that the management activity control of the development of products and assets components is not lost.

![Diagram of product line engineering](image)

**Figure 2.2:** Software product line abstract scheme

### 2.1.3 Architecture

Over recent years there has been a tremendous evolution in model-driven engineering (MDE) which software product lines architectures have started to embrace in order to replace traditional approach originated in the procedure previously defined in section 2.1.2. Nevertheless, there is an elemental architecture which is shared by most frameworks, that is also originated in fundamental software engineering theorems, principles and methodologies.

Such abstract architecture, depicted in figure 2.3 as a process flow of phases, is divided into two major engineering cycles that feedback each other:

- **Domain engineering**
  During this cycle, the commonality and variability of the software product line is defined. Common reusable input assets (software components, methods, models, policies, ...) are grouped and classified. This cycle is responsible of the product line's domain's, but it also is in charge of the design of those assets which will be the key of product line's variability.

  The domain engineering phase in figure 2.3 depicts the activities which are executed, starting with domain requirements elicitation (or simply product line requirements elicitation), to end with the set of reusable input assets. Note that the illustration is adapted to exemplify the architecture when using feature-oriented domain engineering; both feature analysis and modelling will be explained further in section 2.2.

- **Application engineering**
  During this cycle, common input assets (identified in the previous cycle) together with variable input assets are used to create a product as specified by the customer and/or the end user.

---

7 Many authors use the word “framework” instead; however, the author of the thesis refuses to use such concept in this context because the word “framework” not only involves an architecture but many other components, and those will not be explained in this thesis for space and time reasons.

8 For this segregation’s reason, software product line engineering is a model candidate for Model-Driven Engineering development.
This cycle is responsible for deriving product given a set of specified functionalities or needs and binding
the common and variable assets \cite{70}.

The application engineering phase in figure \ref{fig:process_flow} depicts the activities which are executed, starting with the
specific product needs, to end with the manufacture of the end product. Note that the this second cycle is
also adapted to exemplify the architecture when using feature-oriented application engineering; both feature
configuration and derivation of a configuration will be explained further in section \ref{sec:feature_configuration}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{process_flow.png}
\caption{Process flow of the phases in an (object-oriented) software product line stemmed from \cite{12}}
\end{figure}

\subsection{Development process}

Krueger classifies the software product line development process into four\footnote{Check appendix B for the diagrams illustrating each described process} types or categories \cite{49}, where
the proactive and reactive approaches are the most common, and the reactive one is the most popular among
developers.

\begin{itemize}
\item **Proactive**

All development steps of the product line (analysis, architecture and design) are taken in such a way that all
variability is considered for the product line goal domain.

This kind of strategy aims for a fixed number of products because all variation points in the feature model
are known, hence all possible feature combinations can be calculated and it may result in better deployment
times. On the contrary, it is not possible to augment the domain scope, or it is too costly.

\item **Reactive**

The product line is organised in such a way that its design is extended every time there is necessary in a
manner that this extension is feed back to the product line augmenting its domain scope.

This strategy is usually put into practice when there is a set of known requirements and needs for a product
line, but it is also desired that the variability of the product line can be expanded.

As previously mentioned, this approach is the most common among product line engineers because it is the closest to software reuse practices where there are iterative and incremental iterations such as the framework processes derived from the well known Unified Process [33,47].

- Extractive

When there is a collection of products and there is also a need to build a product line comprising the domain scope that these products define, it follows an analysis on these products to retrieve the variation points and the commonalities among them.

This strategy maps an instantiation of a variation point (product) to a part of the feature model. In other words, most of the products that the desired product line can generate are already created; therefore, the process of defining the product line involves extracting the requirements and variation points from the set of available products.

- Refactorive

Although a product line might be properly designed for reuse and unforeseen requirements, new needs or demands suggest that it is much more reasonable to refactor as needed rather than augmenting the variability of the product line.

This strategy is convenient when there is an impression that the product line is evolving. But when taking this approach the designers should take into account all consideration because there may result a partial refactorisation rather than a complete product line architecture refactorisation.

2.1.5 Benefits

Krueger et al. explain in two practical use cases [50,51] how were the process’ experiences of transitioning from the customised software development methodology to a software product line engineering approach. Among the experiences and the difficulties they had to overcome, Krueger et al. indicate some convenient advantages in [48,49], of which:

- Maintenance reduction costs [50,51]

Many developments have found that the code base of the project has achieved a major reduction in size due to removal of duplicated code, which otherwise would have had to be maintained twice, at least.

- Growth in product offer [48]

Some developments have profited from reusable application components that led to a growth of the overall product offer of the development process.

Boeing, is an example of how a layered architecture combined with product line methodologies resulted in a configurable software product line for one of the critical systems, the avionics system software [78]; but also Nokia [11], Siemens or Philips [84], have enhanced their processes as well, and even academia [52].

- Stability in development teams [84]

It seems that staff working in a product line tends to stay longer rather than other people working in another development sections. Since the efficiency of development is higher, and the effort for maintenance is “reused” over time, the satisfaction among developers is higher.

2.1.6 Quality

Among other advantages [69], it is appropriate and of the utmost importance alluding to quality aspects of the engineering process because development teams, businesses and companies embracing this philosophy require certain assurances that the investment will not only result in some of the benefits previously mentioned, but also in a (fact-based) perception that generating products with a software product line will mean more robust end-products,
increased performance, more efficiency,...

There are also methods which help with the process of capturing domain’s requirements and also their management in order to make the process of quality assessment easier [27] to create a “quality-aware” software product line, but while they are not standard, Capability Mature Models (CMM) are an industry standard.

Capability Mature Models are a “collection of best practices to improve, support and help with the organisation’s performance and its processes, often used to assist in the process of improvement across a project, a division, or an entire organisation” [15, 92].

The CMM frameworks can indeed help software product line development practices. While it is not the “enchanting” recipe to improve the overall quality because CMM do suggest what has to be done, but not how, Jones and Soule believes [39] that a maturity and capability level 2 can be achieved.
2.2 Variability in software product lines

Variability management in product lines is a key activity during the life cycle of any product line because not only affects the success of the product line but also the future of companies adopting this emerging area in software engineering [82]. In fact, variability management is the critical characteristic that differentiates software product line development approach from other software development processes [75].

During the last years [10], variability has posed a challenge for software industry development as a consequence of the rapid and sudden necessity of automatising reuse of software components.

According to Bachmann and Clements in [3], variability is the “ability of a system, an asset, or a development environment to support the production of a set of artefacts that differ from each other in a preplanned fashion”, and Schmid and John define variability management in [75] as the process that “encompasses the activities of explicitly representing variability in software artefacts throughout the lifecycle, managing dependences among different variabilities, and supporting the instantiation of the variabilities”.

Meister et al. consider in [58] these variability types (perhaps not with this categorisation, but a similar one) which are of interest to understand that variability affects all stakeholders of a project; hence, a correct administration of these should be implemented:

- **Functional variability**
  Variability visible to customers (set of products with different features)

- **Development variability**
  Variability relevant to developers (modifiability, extensibility and maintainability)

- **Operational variability**
  Variability related to product deployment (performance, scalability, safety,...)

A successful management of variability in software product lines should lead to end products of higher quality. Flexibility increases the number of potential users because they will have the possibility of customising the software they require in line with their needs.

2.2.1 Origins of variability management

Variability was innovatively re-engineered in 1990, when the Software Engineering Institute published the Feature-Oriented Domain Analysis by Kang et al. in [40], where feature modelling (language) was the foremost novelty.

Until then, flow diagrams had been widely used (and popular) among software engineers, designers, developers, testers.... however, these diagrams were, sometimes, confusing because they could not be used with end users because of the target users (for instance, it was common having function names) which were supposed to read them.

Many books about requirement engineering [38, 46] have always cited studies about project failure rates to justify the importance of the initial phase of any project, requirements elicitation.

For that reason, feature models try involving end users and customers mostly (other stakeholders usually have enough technical knowledge to understand most requirements and functionalities diagrams regardless of the “language”) into software development process.

Using a graphical diagram which is understood by all stakeholders, no matter their expertise (indubitably with some guidelines).
Feature modelling is one of the steps of feature-oriented domain analysis process, which is composed of three major stages:

- **Context analysis**
  Defines the scope of the product line’s domain. What will be considered and what not, that is, the domain that the products generated by the product line will be of use to.

- **Domain modelling**
  Defines the commonality and variability among the requirements, functionalities and needs found during the context analysis. And it is the variability what will differentiate one software product from another, but both will be sharing the common elements. This phase is subdivided into three as follows:
  - **Entity-Relationship Modelling (ERM)**
    Knowledge and requirements about applications within the domain are acquired during this subphase, and it is modelled as an entity-relation model.
  - **Feature Analysis (FA)**
    Functionalities that an end user would need (within product line’s domain) are gathered during this subphase and modelled as a feature model.
    An example of a feature model illustrating the functionalities that a management could have to accomplish the task of keeping a minutes’ book is depicted in figure 2.4.

    ![Feature diagram in FODA notation extracted from [74]](image)

  - **Functional Analysis (FuA)**
    Data flows and other exchanges of information between the requirements and characteristics of the domain are identified during this subphase.
    Common functionalities and entity-relation models are the basis of the (abstract) functional model that is produced during this subphase too.

- **Architecture modelling**
  Design of the architecture indispensable to be able to implement the products is carried out during this phase. FODA applies the methodology, Design Approach for Real-Time Systems (DARTS).

### 2.2.2 Feature-based modelling

Feature modelling is a methodology (technique or procedure) used to identify, capture and manage the commonalities and variabilities in system families and product lines during each stage of product line development process [22]. It has many applications such as generative programming (automating application engineering and production based on system families), domain analysis (also known as product line analysis), product line scoping (which features will be supported by a software product line and which ones will not be implemented), and
feature-based product specification \[21\].

But it is the requirement elicitation and analysis stage of any software product development, the two most well-known properties which feature modelling is utilised for, besides becoming the key “player” in the (automated) configuration phase of any software product line.

The term feature is defined by \textit{Kang et al.} in \[40\] as “any prominent and distinctive aspect or characteristic that is visible to various users” although \textit{Czarnecki and Eisenecker} argue in \[20\] that the “visibility” nature of the definition is not exclusive to some users but to anyone involved in the development process of the product line, such as end users, customers, analysts, usability experts, domain engineers, software designers, developers, testers,... and their definition is “a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family”.

As a result, a feature may represent a functional or non-functional requirement, a characteristic of the platform where the end product shall be deployed,... These features, are organised in what is called, feature diagram, a tree whose root node symbolises the concept that the feature model is characterising, and each root’s child node is a feature, either directly or indirectly (thus, a feature may be a child of another feature).

Some features will be common to all products of the product line modelled in the feature diagram, but some other features will only exist in some products, that is, for some configurations of the feature diagram. There are also groups of features which are related between them because the semantic information that these features are portraying requires a categorisation. These two main types of features (common and optional, and grouped features) are the lexicon which is used by FODA.

Features are organised in a hierarchical composition mainly because the underlying structure is a tree\[10\] but this is a consequence of the methodology which introduced feature modelling, FODA, which in turn is a consequence of traditional domain analysis (a feature can be considered as a module that can be decomposed in submodules).

A feature model is a collection of feature diagrams and additional information such as feature descriptions, constraints, requirements that were not necessary to be modelled as features, libraries,... and any kind of information that is relevant to any stakeholder involved in the development of the product line.

Figure 2.5 depicts a feature diagram illustrating the concept of a search engine. But there are a couple of constraints to be considered in this example (in the opinion of \textit{Mendonça}, author of the example) that can not be modelled as features and/or hierarchical relations between features:

- **search by language requires page translation**
  
  If a stakeholder wants a search engine with the functionality of searching by language, the feature model automatically adds the functionality of translating a page

- **page preview excludes svg**
  
  If a stakeholder wants a search engine with the functionality of previewing pages, the feature model will not allow previewing pages whose document type is a ‘svg’ file.

Given these constraints, it is easy to understand that modelling them as relations would not be possible because the feature model “language” does not have any syntax nor lexicon which these constraints can be expressed with, although the notation used in this example has a couple of elements more than the original notation from FODA (next subsection will explain the notation used in the example and an extension to it which is more explicit).

\[10\] Later on, the reader will have the opportunity to discover that another complex structures can be used instead, for instance, graphs or hyper-graphs
Analysis process

FODA [40] defines a full domain analysis methodology which was briefly described in section 2.2.1, but the procedures and steps employed for feature diagram’s construction are valuable considering the role that feature modelling plays in (model-driven) software product line engineering; hence, the reason to detail the major steps.

- **Documentation**
  In similar fashion than in an “individualised” software development process, any source of information is useful to gather all details about the product line’s domain scope, among them:

  - **Domain experts**
    These are the personnel who is more likely to know what to look for, how and when. However, it might as well happen that more professionals are needed because each one is a specialist in a particular area of expertise.

  - **Requirement engineers**
    These analysts are usually responsible for requirement elicitation, and they are often close to those stakeholders which are on the product side, that is, end users. They habitually interview end users and analyse users’ comments, suggestions, answers,... to elicit requirements of interest for the domain.

  - **Related applications**
    Studying, reviewing and analysing current applications within the scope of the domain or close to it is singularly advantageous for the documentation because there is plenty of available documentation and some others details such as the architecture, or deployment guidelines, can solely be found in other software. However, the cost of this task is high because it requires time, which is a scarce resource.

  - **Standards and specifications**
    Valid reference for architectonical matters or any other problem that requires compliance with an industrial framework or specification.

  - **Literature**
    Books, scientific papers,... are a good source of theories, methodologies, procedures, techniques, models,... but these should not be the main source of information because they deal with a very small percentage of domain scope.

- **Feature classification and identification**

  Kang et al. proposed a categorisation (open for changes) of the features in [40]:

  - **Capabilities**
2.2. Variability in Software Product Lines

- Functional
- Operational
- User interface
  - Operating environments
  - Domain technology
  - Implementation techniques

Once the categorisation has been accomplished (it may occur that some features are discarded because they are not related to the domain), a descriptive name should be assigned to each identified feature.

**Model design**

With all features categorised and named, the diagram can be designed. The intention is to create a hierarchical map of the features with the help of the existing relationships between these features, and taking into account if a feature is common to any product or if there is a possibility that a product does not implement it, and if a feature is related to other features when the relation is not hierarchical (hence, it will be a feature in a group).

But the model will not be complete with the feature diagram. During diagram’s design, there might be some relationships which was not viable to model; hence, they can be modelled as textual (or graphical) constraints. And a description of what does each feature will also be required to have a complete model (for which, the previous step of documentation is quite useful not only to find the requirements and functionalities of the product line).

**Verification**

In all likelihood, this is the most important step of the analysis process because the model is verified not only to find out about ambiguities or bad implementation of requirements or lack of desired functionalities (informal verification), but also to verify the model formally; that is, a formal verification of the model shall also be carried out.

Informal verification is conducted with any stakeholder who has not been participated in any analysis step to minimise any influence or deviation due to the “extra” knowledge. The intention is that anyone who has not seen the feature model before should be able to understand its meaning and possibly extend it further if needed.

Formal verification is conducted by means of formal methods from graph theory. Feature diagram’s underlying structure is a tree (or a graph); hence, it should obey the definition of a tree (or a graph) along with other formalisms with respect to the considerations exclusive to feature modelling (for instance, there can not be any orphan feature in a feature diagram).

**Notation**

Notation used in feature diagrams has not changed much since they were first proposed by Kang et al.. This first proposal, for which figure 2.4 illustrates the concept of a “minutes’ book”, used a simple but effective lexicon.

Features (nodes) are related between each other with lines (edges); for instance, date/time feature and the root feature, minutes’ book. Features that are optional have an ending empty circle at the end of the edge which links them to their corresponding parent; for instance, agenda is an optional feature. Any other feature (excluding those whose edges are part of an arc, such as convention, directorate and commission) is mandatory, that is,

11 Some authors use concept to identify the root feature of a feature model because it represents the feature model, but some others do not do this consideration.
it is common to any product generated by the product line. The last kind of relationship is the alternative decomposition arc, *XOR*, which functions like the *XOR* boolean logical operation (at most one feature can be selected regardless of the number of features); any feature under an alternative relationship will be selected if the other features under the same arc are not selected; for instance, *convention, directorate, and commission* features.

FODA notation has a “flaw”, it does not have an *OR* decomposition arc; for instance, in figure 2.4 it was not viable selecting *directorate* and *commission* features at the same time, even though it may happen (a commission reports back to the directorate during a meeting where both are present).

This weakness was solved by Czarnecki and Eisenecker in [20] when they introduced the *OR* decomposition and more graphical lexicon. An example of this (popular) notation is illustrated in figure 2.5.

The first novelty is the boxing of features which may seem unimportant, but it eases the task of reading the diagram. Mandatory features have a filled circle at the end of the edge which relates the child feature with its parent; for instance, *document type* feature. And the new *OR* decomposition arc, which functions like the *OR* boolean logical operation (at least one feature must be selected, and at most all of them), is depicted likewise a *XOR* decomposition arc, but the arc is filled instead; for instance, all child features of *search by language* parent feature: *portuguese, spanish*, and *english*. Moreover, observe that features under *OR* or *XOR* decomposition arcs have an empty circle at the end of the feature.

At a later date, Czarnecki et al. augmented in [21, 22] their former lexicon, and proposed a cardinality-based feature diagram among other novelty which will be explained further in the next subsection, staged configuration.

Cardinality was introduced to eliminate ambiguities that previous notations could lead to. The graphical notation changes so that all features have an empty circle at the end of the edge that connects them to their corresponding parent, and all arcs are empty as well; therefore, a visual inspection of the diagram will not provide any information but it will even confuse because it may seem that either features are optional (those which are not under any arc), or features belong to *XOR* decomposition arcs. For this reason, the cardinality is the base of the diagram, without this extra information, the feature diagram is meaningless.

Czarnecki et al. outlined the following feature cardinality classification:

- **Feature cardinality**
  These are applied to solitary features, and expressed as follows, \( [m \ldots n] / m, n \in \mathbb{Z}^+ \land m \geq 0, n > 0, n \geq m \)
  where \( m \) and \( n \) are the minimum and maximum cardinalities respectively.
  Therefore, when a solitary feature has a cardinality of \([1 \ldots 1]\) or \([0 \ldots 1]\), it is either a mandatory or an optional feature respectively.
  Semantic meaning of any other cardinality combination is “cloning”, the solitary feature may be repeated in any configuration at least \( m \) times and at most \( n \) times.

- **Group cardinality**
  These are applied to grouped features, and expressed as follows, \( <m \ldots n> / m, n \in \mathbb{Z}^+ \land m \geq 0, n > 0, n \geq m \)
  where \( m \) and \( n \) are the minimum and maximum cardinalities respectively.
  Semantic meaning of group cardinalities is slightly different from feature cardinalities, the cardinality influences the arc which groups a set of features and not each feature. Therefore, any arc must have, for any configuration, selected at least \( m \) of its features and at most \( n \) of them.
  It is evident that maximum cardinality in a group cardinality can not be more than the total number of features.
2.2. Variability in Software Product Lines

tures under the arc. Sometimes, authors use $n = \ast$ to indicate that maximum number of features in the arc without explicitly expressing the number.

Figure 2.6 exemplifies the extended notation of previous figure 2.5. Since the transformation from one notation to the other (and vice-versa) is straightforward, and the feature diagram of the depicted concept is the same; hence, by visual inspection the reader can compare both illustrations; no more explanatory comments will be provided.

There are more notations in scientific literature; for instance, [Metzger and Heymans] wrote a technical report comparing some feature diagram examples that they found in research literature from diverse authors.

Finally, an original meta-model of feature modelling is included in [Vranic]. The meta-model proposed by Vranic uses the non-extended Czarnecki-Eisenecker notation and illustrates all elements that are required to have a complete feature model (that is, feature diagrams, descriptive documentation and textual or graphical constraints).

Configuration

Configuration is the process of deriving a configuration, selecting or “removing” features, from the feature diagram (while taking any constraint into account) to reduce the variability that the feature model is depicting. The outcome of the configuration can be either a concrete configuration which uniquely identifies a product in the product line (because there is no point of variability) or a partial configuration which represents the variability of a subset of products in the product line; that is, a partial configuration is an specialisation because it yields another feature diagram.

Partial configurations are the second novelty that [Czarnecki et al.] proposed in [21, 22], although the name used by the authors is staged configuration.

[Czarnecki et al.] define, in [22], staged configuration as the process where “each stage takes a feature model and yields a specialised feature model, where the set of systems described by the specialised model is a subset of the systems described by the feature model to be specialised” and in [21] they considered as well, for some product contexts, different feature models for each stage, [Czarnecki et al.] named it: “multi-level configurations”.

Therefore, there are two configuration procedures relying on how it is carried out:

• **Configuration**
  A complete and unique configuration is derived from the feature diagram.

• **Specialisation**
Chapter 2. Background and Methodology

A sequence of partial (staged) configurations are derived, each one from the previous one, starting with the product line’s feature diagram, and then a configuration is derived from the last staged configuration, which is a fully specialised feature diagram.

Czarnecki et al. proposed in [22] a guide of specialisation steps, because they believe specialisation is carried out as an ordered list of steps. Those steps are listed below:

- **Refining feature cardinalities**
  Given an interval \([m...n] / m, n \in \mathbb{Z}^+ \land m \geq 0, n > 0, n \geq m\), refine the interval by increasing \(m\) or decreasing \(n\) as long as \(n \geq m\) holds.
  If \(n = \times\) in the original interval, it can be substituted with a number as long as the previous property holds. A trivial case would be \([0...0]\), that is \(m = n = 0\), which has the semantic meaning of removing the feature (and the tree below it).

- **Refining group cardinalities**
  Given an interval \(<m-n> / m, n \in \mathbb{Z}^+ \land m \geq 0, n > 0, n \geq m\), refine the interval by increasing \(m\) or decreasing \(n\) as long as \(n \geq m\) holds.

- **Removing a subfeature from a group**
  Given a group featured arc with group cardinality, \(<m-n> / m, n \in \mathbb{Z}^+ \land m \geq 0, n > 0, n \geq m\) and group size \(k\), refine the group by removing one subfeature, at a time, as long as \(m < k\) holds (new group size would be \(k-1\) and new group cardinality would be \(<m-min(n,k-1)>\).

- **Selecting a subfeature from a group**
  Given a group featured arc with group cardinality, \(<m-n> / m, n \in \mathbb{Z}^+ \land m \geq 0, n > 0, n \geq m\) and group size \(k\), refine the group by selecting one subfeature, at a time, as long as \(n > 0\) holds (new group size would be \(k-1\) and new group cardinality would be \(<max(0,m-1)-(n-1)>\).

- **Assigning an attribute value**
  Assign a value to an uninitialised attribute.

- **Cloning a solitary subfeature**
  Given a solitary feature with a feature cardinality which allows cloning, this step clones the feature and its subtree.

In figure 2.7, an example (extracted from [54]) of a feature diagram and two specialisations is shown. The first staged configuration has chosen to remove the registered feature (or reduce the solitary feature cardinality to \([0...0]\)), while the second staged configuration, based on the first staged configuration, has chosen credit card feature over emoney feature (emoney feature has been removed from a grouped feature, and the group cardinality has been adjusted accordingly), and is a fully specified feature diagram, hence a unique configuration.

---

14 Czarnecki et al. accept split intervals for some cases, for instance, feature cardinalities, but they will not be considered in this text for simplicity.
15 Check eight page in [22] for a formal definition.
2.2. Variability in Software Product Lines

(a) Feature diagram

(b) Feature diagram (extended notation)

(c) First staged configuration

(d) Second staged configuration

Figure 2.7: Staged configuration of feature diagram
CHAPTER 2. BACKGROUND AND METHODOLOGY

2.3 Directed hyper-graphs modelling paradigm

Foundations of graph theory are believed to have been laid by Leonhard Euler when in 1736 he published the paper "Solutio problematis ad geometriam situs pertinentis" also known as the "Königsberg Bridge Problem" [8]. Although it was not until a century and a half later when the concept of a graph was effectively introduced by Sylvester in the paper published in the scientific magazine "Nature" [83].

Graphs have become one of the discrete mathematical research areas that computer science has studied, analysed and explored most, extensively used and successfully applied from the very beginning since its existence, because of its properties and parallelisms with varying data structure representations employed in many computer science problems, such as finite state machines.

Graphs are of extremely importance in computer science as a consequence of their ability to represent information which would not be possible otherwise because it will lead to an abstract depiction of the problem while the graph representation itself would exhibit new strategies or techniques to solve the problem, and sometimes procure the answer (partial, or even complete) to the problem.

2.3.1 Graph definition

Intuitively [16], a graph depicts a set of connections or relations (edges or arcs) between objects (nodes or vertices), where those relations may be asymmetric or directed (the flow of information between two objects is established with a source and a destination) or, symmetric or undirected (the information can circulate in both directions between two objects; that is, an object might be the source of the relation at a time, and later it may become the destination of the flow of information in the same relation).

Hence, a more formal definition [91] of a graph $G$ is a pair of a finite set of nodes $V$ and a set of edges $E$; that is to say, $G = (V, E)$, where $E = \{(u, v) / u, v \in V\}$.

In the "Königsberg Bridge Problem", the goal was being able to walk through the city of Königsberg (starting in one place and ending in the same or another one) so that each bridge was crossed once, and only once (each bridge had to be crossed completely). The problem is depicted in figure 2.8, representing both the map of the city and the corresponding view as a graph (where the lands are symbolised as nodes, identified with a letter; and the bridges are symbolised as edges, identified with a number).

![Figure 2.8: Euler's Königsberg graph problem [8] (a) Map of Königsberg's bridges (b) Equivalent graph of Königsberg's bridges](image)

16This definition aims for an easy understanding of what a graph is; therefore, many assumptions have to be made, but a precise definition will follow afterwards.
However, Euler found out that this problem had no solution, that is, there was not any path in which each bridge was crossed just once. Every time someone wanted to visit and leave a part of the city, two bridges had to be crossed (one to get in, and another one to get out); therefore, the number of bridges having one side on each land should always be even, but in Königsberg this number was odd for both sides of the river and the two islands. This example describes the first graph problem in mathematics history; even though it was a problem with no solution, Euler’s abstract model of Königsberg was crucial for the future of graph theory; in fact, his results also introduced the concepts of path and traversal.

### 2.3.2 Undirected hyper-graphs

Formally \[ \mathcal{H}, \] a hyper-graph \[ \mathcal{H} \] is a pair consisting of a vertex set \( V \) and a hyper-edge set \( E \); that is, \( \mathcal{H} = (V, E) \), where,

- \( V = \{v_1, v_2, v_3, \ldots, v_n\} \) is the finite set of vertices (or nodes)
- \( E = \{E_1, E_2, E_3, \ldots, E_m\} / E_i \subseteq V \forall i = 1, \ldots, m \) is the set of hyper-edges (or hyper-arcs)

and the hyper-edge set \( E \) meets the following two properties,

- \( E_i \neq \emptyset \forall i = 1, \ldots, m \) (Empty hyper-edges are not allowed)
- \( \bigcup_{i=1}^{m} E_i = V \) (The set of vertices of all hyper-edges is the vertex set)

The formal definition of a hyper-graph generalises the definition of a graph. Therefore, a graph can be defined in terms of a hyper-graph (when the cardinality of all hyper-edges from the \( E \) set is two, and there are no isolated vertices) as follows: \( |E_i| = 2 \forall i = 1, \ldots, m \).

Although the size of a graph is proportional to the size of its sets of vertices and edges, the size of a hyper-graph depends on the cardinality of its hyper-edges as well; that is, the size of a hyper-graph is the sum of all cardinalities from its hyper-edges: size\( (\mathcal{H}) = \sum_{E_i \in E} |E_i| \forall i = 1, \ldots, m \).

Biological networks is an example of undirected hyper-graphs, although they are often represented as graphs. For instance, a pair of proteins interacting with each other results in what is called a protein complex (in figure 2.9, proteins A and E, or D and F); but the complexity of some associations surpasses the modelling capabilities of graphs (in figure 2.9, proteins A, B, C and D); however, a hyper-graph overcomes those representation difficulties.

![Hyper-graph of network](image)

![Corresponding graph of network's hyper-graph](image)

**Figure 2.9:** Hyper-graph and graph of a protein-protein network interaction based on [44]

---

17 Even though the term should have the “undirected” preceding word, it has purposely been omitted in order to ease the reading. However, there are significant differences between undirected and directed hyper-graphs which will be described in the following subsection, but given that an undirected hyper-graph is the simplest case, the adjective will be omitted for this subsection.

18 The figure is a fictitious work derived from the example hyper-graph usage in cellular networks published in [44].
Even though right subfigure in 2.9 is representing the equivalent graph of the protein to protein interaction hyper-graph on the left subfigure, there is not a loss of information as the set (also named hyper-edge in this context) of four proteins, A, B, C and D, from the hyper-graph, is modelled as six sets of two proteins each; that is, all combinations among the proteins composing the protein complex number four. Note, that protein D is the sole member of complex number five, which is depicted as a hyper-edge (or an edge in the equivalent graph) associated to itself.

2.3.3 Directed hyper-graphs

A directed hyper-graph[19], or simply hyper-graph [30], is an ordered pair consisting of a vertex set \( \mathcal{V} \) and a hyper-edge set \( \mathcal{E} \); that is, a hyper-graph with directed hyper-edges, \( \mathcal{H} = (\mathcal{V}, \mathcal{E}) \), where:

- \( \mathcal{V} = \{v_1, v_2, v_3, \ldots, v_n\} \) is the finite set of vertices (or nodes)
- \( \mathcal{E} = \{E_1, E_2, E_3, \ldots, E_m\} / E_i \subseteq \mathcal{V} \forall i = 1, \ldots, m \) is the set of directed hyper-edges (or hyper-arcs)
- \( E_i = (T(E_i), H(E_i)) / T(E_i) \subseteq \mathcal{V} \land H(E_i) \subseteq \mathcal{V} - T(E_i) \) is a directed hyper-edge where \( T(E_i) \) is the set of tail nodes and \( H(E_i) \) is the set of head nodes of \( E_i \)

A hyper-graph with one or more hyper-arcs whose head set is empty, \( E_i = (T(E_i), \emptyset) \), or whose tail set is empty, \( E_i = (\emptyset, H(E_i)) \) is valid. However, hyper-graphs with hyper-arcs where both head and tail sets are empty, \( E_i = (\emptyset, \emptyset) \), are not allowed; moreover, it would not add any information to the hyper-graph.

The following figure 2.10 illustrates an example of a directed hyper-graph. Note that there is a hyper-arc, number six, with an empty head set, and that the example is a cycle-free hyper-graph (there are no paths[20] following the direction of the arcs’ arrows, starting and ending in the same node).

![Figure 2.10: Directed hyper-graph](image)

When considering a directed hyper-arc, hyper-arcs may be classified into two categories according to the cardinality (or size) of the source and tail sets.

This kind of hyper-arcs is of significant relevance for many problems, because the constraint imposed on the source or the tail sets facilitates the modelling and/or the management of those problems where this approach is suitable.

A hyper-arc \( E = (T(E), H(E)) \),

---

19 As previously pointed, from now on, the term hyper-graph will be used instead of directed hyper-graph following the “convention” of most of the relevant literature for the research field of interest in this thesis. But the reader should be careful when reading any article, book,... in case there is no distinction; even though the directed approach is habitually the most used given its properties.

20 Paths on hyper-graphs will be explained in “Paths and hyper-paths” subsection of 2.3.3
2.3. DIRECTED HYPER-GRAPHS MODELLING PARADIGM

• is a backward hyper-arc (or simply B-arc) if and only if \(|H(E)| = 1\) (The hyper-arc only has one node as its head)

• is a forward hyper-arc (or simply F-arc) if and only if \(|T(E)| = 1\) (The hyper-arc only has one node as its tail)

Following figure 2.11 depicts the two types of hyper-arcs that have been defined. Note that the cardinality set, either the source or the tail, in both hyper-arcs, is one.

![Directed hyper-arcs](image)

**Figure 2.11:** Types of directed hyper-arcs

According to the number of incoming or outgoing hyper-arcs for any node \(v \in V\) given a hyper-graph \(H = (V, E)\), [27] denote in \([30, 31]\) by,

- the forward star of node \(v\): \(FS(v) = \{E_i \in E / v \in T(E_i) \forall i = 1, \ldots, m\}\) (The set of hyper-arcs which have the node \(v\) as a member of their tail set)

- the backward star of node \(v\): \(BS(v) = \{E_i \in E / v \in H(E_i) \forall i = 1, \ldots, m\}\) (The set of hyper-arcs which have the node \(v\) as a member of their head set)

For instance, the forward star of node \(B\) in figure 2.10 would be, \(FS(B) = \{1, 2, 3\}\), while the backward star of node \(E\) would be, \(BS(E) = \{4, 5\}\).

A hyper-graph whose hyper-arcs are B-arcs is named B-hyper-graph (or B-graph) while if its hyper-arcs are F-arcs the name is F-hyper-graph (or F-graph). If the hyper-arcs of a hyper-graph are either B-arcs or F-arcs, then the name is BF-hyper-graph (or BF-graph).

Any hyper-graph \(\mathcal{H}\) can be converted to a BF-graph by replacing each hyper-arc \(E_i\) which is not already a B-arc/F-arc with the hyper-arcs \(E_i' = (T(E_i), \{v_q\})\) and \(E_i'' = (\{v_q\}, H(E_i))\) where \(v_q\) is a new (dummy) node [30, 63] (Each hyper-arc is transformed into a B-arc and a F-arc by means of an extra node).

Figure 2.12 illustrates the above described transformation process (node \(Z\) in right figure would be the introduced dummy node \(v_q\)).

Finally, a hyper-graph \(\tilde{\mathcal{H}} = (\tilde{V}, \tilde{E})\), is a sub-hyper-graph of \(\mathcal{H} = (V, E)\), and thus written \([21]\) as \(\tilde{\mathcal{H}} \subseteq \mathcal{H}\), if and only if, \(\tilde{V} \subseteq V \land \tilde{E} \subseteq E\) [64].

**Paths and hyper-paths**

Considering a hyper-graph \(\mathcal{H} = (V, E)\), a destination node \(t\) is connected to an origin node \(s\) if there is a path \(P_{st}\) in \(\mathcal{H}\); where the path is a sequence of nodes and hyper-arcs, \(P_{st} = \{v_1, E_1, v_2, E_2, v_3, E_3, \ldots, E_q, v_{q+1}\}\), such that \(s = v_1 \land t = v_{q+1}, s \in T(E_1) \land t \in H(E_q)\) and \(v_j \in H(E_{j-1}) \cap T(E_j) \forall j = 2, \ldots, q\).

\[21\] It is likely that the reader also finds an alternative, but equally valid, definition, “\(\tilde{\mathcal{H}}\) is contained in \(\mathcal{H}\)”
When \( t \in T(E_1) \), the path \( P_s \) is a cycle (a trivial case would be when, the origin and destination nodes are the same, \( t = s \)). A path is not a cycle when it does not contain any subpath which is a cycle \(^{[30]}\); for instance, \( v_i \in T(E_j) \Rightarrow j \geq i \land 1 \leq i \leq (q + 1) \) \(^{[65]}\). If a hyper-graph \( H \) does not contain any cycle, it is said to be an acyclic hyper-graph.

An example of a path connecting destination node \( G \) with origin node \( F \) is depicted in figure 2.13 with a thicker black line. Note that the path \( P'_{FG} \) has two alternatives, \( P'_{FG} = \{ F, 4, A, 1, D, 3, G \} \) and \( P''_{FG} = \{ F, 4, A, 1, I, 2, D, 3, G \} \); however, the former is the shortest path between \( F \) and \( G \).

Gallo and Scutella define in \(^{[32]}\) a directed hyper-path \(^{[22]}P_{\mathcal{R}, t}\) from a root set \( \mathcal{R} \subseteq \mathcal{V} \) to a destination node \( t \in \mathcal{V} \) in \( H \) as the minimal acyclic sub-hyper-graph of \( H \) (containing the set of nodes \( \mathcal{R} \cup \{ t \} \), such that each node has one incoming hyper-arc, excluding the nodes of \( \mathcal{R} \)).

Derived from the previous definition, Gallo and Scutella define a directed hyper-tree \( T_{\mathcal{R}} \) whose root is \( \mathcal{R} \) in \( H \) as an acyclic sub-hyper-graph of \( H \) (containing the set of nodes \( \mathcal{R} \), such that each node has one incoming hyper-arc, excluding the nodes of \( \mathcal{R} \)).

Given a hyper-graph \( H = (\mathcal{V}, \mathcal{E}) \), Gallo et al. define \(^{[30]}\) in \(^{[32]}\) a B-hyper-path (or simply B-path) \( \pi'_{st} \), with origin node \( s \) and destination node \( t \), as the minimal hyper-graph \( H'_{\pi} = (\mathcal{V}', \mathcal{E}') \) which satisfies the following

\(^{22}\) Once again, it will be assumed from now on that any hyper-path is directed unless specified otherwise

\(^{32}\) Nguyen et al. similarly define a hyper-path in \(^{[63]}\)
2.3. DIRECTED HYPER-GRAPHS MODELLING PARADIGM

conditions,

- \( E_\pi \subseteq E \)

- \( s, t \in V_\pi \land V_\pi = \bigcup_{E_i \in E_\pi}(T(E_i) \cup H(E_i)) \)

- \( v \in V_\pi - \{s\} \Rightarrow v \) is connected to \( s \) in \( H_\pi \) by means of a cycle-free simple path

Analogously, a F-hyper-path (or simply F-path) \( \pi^{''}_{st} \), with origin node \( s \) and destination node \( t \), is defined as the symmetric image of a B-path with origin node \( t \) and destination node \( s \). And a BF-hyper-path (or simply BF-path) \( \pi^{'''}_{st} \), with origin node \( s \) and destination node \( t \), is defined as the hyper-graph which is both a B-path and a F-path.

Gallo et al. also define the a node \( u \) is B-connected (or F-connected or BF-connected) to another node \( v \) if there is a B-path (or F-path or BF-path) \( \pi_{uv} \) in hyper-graph \( H \). Therefore, given a hyper-arc \( E = (T(E), H(E)) \), a B-reduction\(^{24} \) of \( E \) is the B-arc \( E_B = (T(E_B), \{v\}) \) such that \( T(E) = T(E_B) \) and \( v = H(E) \).

However, Nielsen and Pretolani have pointed out in [65] that previous definition of a B-hyper-path (hence, considering B-hyper-graphs) is not correct unless it is imposed that path \( \pi_{st} \) must be acyclic. Nielsen and Pretolani have manifested that a valid definition of a B-path has yet to be defined. Later on, Nielsen opted for the definition [2] of Ausiello et al. in his thesis [64], which follows.

A hyper-path \( \pi_{st} = (V_\pi, E_\pi) \), with origin node \( s \) and destination node \( t \), is the minimal sub-hyper-graph of \( H \) which satisfies,

- if \( t = s \) \( \Rightarrow E_\pi = \emptyset \)

- if \( t \neq s \) \( \Rightarrow q \geq 1 \) hyper-arcs in \( E_\pi \) can be ordered in a sequence \((E_1, E_2, \ldots, E_q)\) such that,
  - \( t = H(E_q) \)
  - \( T(E_i) \subseteq \{s\} \cup \{H(E_1), H(E_2), \ldots, H(E_{i-1})\} \forall E_i \in E_\pi \)
  - No proper sub-hyper-graph of \( \pi_{st} \) is an \( s - t \) hyper-path

These differing definitions and opinions on basic concepts of hyper-graph theory are a small sample of what the reader can find in available literature, but it is important to realise that there are diverse research trends on hyper-graph theory. Nevertheless, the definitions previously explained are the ones that will be used in this thesis, especially those from Gallo et al.

2.3.4 Applications

Previously, undirected hyper-graphs were explained with an example (check figure 2.9) of a biological entity, a protein to protein complex, could be modelled as a graph, but also as an undirected hyper-graph. That example is a straightforward representation of a combinatorial problem that was better represented as a (undirected) hyper-graph rather than a (simple) graph. But there are other problems that are not feasible to be represented as graphs without losing information.

Hyper-graphs are merely a name for a (usually visual) representation of a combinatorial problem [90]. Such problems have been studied for decades because they arise in many mathematical research areas [35] such as algebra or topology but also in some other related areas such as operations research, optimisation or computer science.

Notwithstanding that there are two types of hyper-graphs, undirected and directed, the former ones have been predominantly used for modelling or structural purposes rather than also for algorithmic purposes [32]. Gallo and Scutella justify in [32] that the foremost reason to use directed hyper-graphs is their ability to effortlessly represent many to one relations which graphs are not suited without supplementary information (for instance, “and-or”
graphs or “labelled” graphs).

Some of those applications where directed hyper-graph modelling is of use are listed below.

- **Transportation** [30, 32, 64]
- **Formal languages** [32]
- **Databases** [30, 32]
- **Traversal algorithms** [2, 30, 63]
- **Production and manufacture** [32]
- **Satisfiability problem solving** [30, 59]
2.4 Work methodology

After introducing the theoretical foundations of this thesis, and practical applications, in previous sections of this chapter, this last section will describe the methodology and the tools used in the following chapter.

- **Implementation of hyper-graph data structure**
  Programming language can be any, but for simplicity, Java could be a good option to start with.

- **Implementation of basic traversal hyper-graph algorithms**
  It is recommended to implement raw traversal algorithms and test them to understand how the data structure and the traversing works [30].

- **Specialisation of hyper-graph data structure to fit feature model characteristics**
  Feature model abstraction should be implemented as an specialisation of the hyper-graph data structure and follow basic object-oriented principles.

- **Customisation of specific traversal algorithms for staged configuration of feature diagrams**
  Customised traversal algorithms, that is verification algorithms, should be implemented using the raw traversal algorithms as a guide.

- **Benchmarking of algorithms and customisations as needed**
  Tests will be carried out using selected models from Software Product Line On-Line Tools (SPLIT) [60]. The measure of interest is time and feature model characteristics as well as user’s configuration (selection).
  It is recommended to use a profiling tool to improve the accuracy of the measures.

- **Analysis of results**
  While analysing results using selected tests it is likely that modifications have to be done on implementation, either for debugging reasons, performance,... (in any case, the reason which triggered the modification should be documented, excluding changes due to bad implementations).

2.4.1 Delimitations

This project will not cover the development of any kind of tool to model, validate, configure nor develop software product lines. However, a tool [25] (or a set of tools) would absolutely benefit from the results of this project. These results should only be suitable algorithms for verification of feature diagrams and any (user) chosen configuration, where the feature diagrams are modelled with an underlying structure of a forward hyper-graph. No tools which may include these algorithms are part of this project.

---

25 Such as the one being developed by GIRO research group
Chapter 3

Feature diagrams and hyper-graphs

“The danger of the past was that men became slaves. The danger of the future is that men may become robots”
Erich Fromm

Representation of variability and commonality among features of a software product line has posed many challenges since the first proposal, Feature Oriented Domain Analysis, made in 1990 by Kang et al. in [40], because even though two decades have passed, there is yet to be defined a common criteria or definition which fully describes the terms of feature and feature diagram (or feature model).

An example of the description of FODA, which was explained in subsection 2.2 was illustrated in figure 2.4. The tree showed the features that a root minutes’ book feature would be decomposed in; for instance, the minute itself or the type of people meeting are decomposed as AND relationships (edges), but the agenda feature is decomposed as an OPTIONAL relationship (edge), while the type of people who were in the meeting is decomposed as a X-OR relationship (edge).

Note, that the latter feature, meeting, ought to be decomposed as an OR relationship (edge) because it is feasible that a commission is meeting with the directorate, hence this would not be possible to be modelled according to FODA’s semantics.

3.1 Tree and graph modelling of feature diagrams

Since the first feature model proposal by Kang et al. in 1990, represented as the well known data structure of a tree, many other proposals have been suggested by scientific community during the last two decades.

The ambition of some proposals usually was dealing with missing semantic expressiveness that some feature model proposals lacked, some others tried creating a new language to avoid being subject to the limitations of the current research.

Despite all these varying proposals, the basics of FODA have continued to exist in mostly all models. The tree data structure is, for the most part, the characteristic that has remained nearly intact. Many proposals continued using a tree while some others tried an upper abstraction level, a graph.

The use of graphs or trees is not uncommon in computer science; in fact, it is almost mandatory to solve many problems, given its demonstrated applications in the field, but it is of significance discovering another practical use of graph theory.

Schobbens et al. presented in [76] a feature model language, named Varied Feature Diagram (VFD), which the authors define as “expressively complete, harmfully irredundant, that can embed all other known variants, and linearly succinct”. This proposal is the result of an extensive comparison analysis to examine four key aspects

1 Some authors use the term “feature model language” because the description of each proposal, in the end, is a language with grammar (a set of rules) and lexicon (the elements to depict the feature model with)
2 The analysis is a follow-up of another one undertook by the same authors in 2004
of the most important proposals of feature diagram languages (modelled as trees or (directed acyclic) graph): expressiveness, embeddability, succinctness and redundancy.

Later on, [62] conducted a practical comparison study of genuine feature model languages’ examples, and they mostly used the same proposals.

For this reason, and as an introduction to feature modelling as graph derivatives, it follows a brief description of most of the feature model languages compared by both Schobbens et al. and Metzger and Heymans, because these proposals are of fundamental interest to any research on feature diagram modelling:

- **Feature Oriented Domain Analysis (FODA)** [40]
  Kang et al. proposal is constructed on a tree data structure. The root node is the sole feature that is required, and it usually describes the abstraction that is modelled as a tree.
  The nodes represent (requirement) features (which can be optional or mandatory), and the edges represent relationships between features (which can be **AND** or **XOR** decompositions). Other types of relationships between features are described as textual constraints (which can be either **require** or **mutex** restrictions).
  Figure 2.4 illustrates an example of Kang et al. proposal, and section 2.2 explains it further.

- **Feature Oriented Reuse Method (FORM)** [41]
  Kang et al. propose an extension of FODA where the data structure is a directed acyclic graph, which is organised in four layers: capability, operating environment, domain technology and implementation technique layer. The capability layer models the product line variability whereas the other layers model the software variability. Layers are inter-connected by means of “implemented-by” edges.
  Besides decomposition (**AND** and **XOR**) relationships, FORM adds generalisation and specialisation relationships (being these two new types the ones which transform the tree into a graph), and the previously mentioned “implemented-by” relation. Constraints are still described as text.

- **Generative Programming (GP)** [24]
  Czarnecki et al. propose a derivative of FODA adapted to the paradigm of “Generative Programming”, hence the underlying data structure of a tree holds. However, in later works from the same authors, [21, 22, 23], some novelties were introduced, such as staged configuration and, group and feature cardinalities.
  These feature models adds, besides the feature and group cardinalities, an **OR** relationship to FODA’s ones. Textual constraint’s characterisation is still inherited from FODA though.
  Both staged configuration and, group and feature cardinalities are explained in section 2.2 given its relevance for the correct understanding of the variability management.

- **Feature Reuse-Driven Software Engineering (FeatuRSEB)** [37]
  Griss et al. propose an amalgamation of FODA and the business oriented and use-case driven methodology, Reuse-Driven Software Engineering Business (RSEB). The underlying data structure of the feature model is a directed acyclic graph.
  An **OR** relationship is added to the **AND** and **XOR** from FODA. FeatureRSEB names “variation points” to **OR** and **XOR** decompositions (being the features part of the decomposition the variants). **require** and **mutex** constraints are inherited from FODA but the representation is graphical (a stereotyped edge).

- **Product Line Use case modelling for System and Software engineering (PLUSS)** [26]
  Eriksson et al. propose a derivative of FeatuRSEB where a combination of feature diagrams and use cases is utilised. However, the underlying data structure is a tree despite the fact that FeatuRSEB uses a graph.

3The reader is more than welcome to read the comparison study that Schobbens et al. completed because this thesis will not detail those results, but the author of this thesis believes the paper groups the most relevant feature diagram proposals in available literature
This feature model language does not add anything to FeatuRSEB although it does not have textual constraints but only graphical (which is expected given its aim of combining with software engineering and use cases).

- [Riebisch et al. 72]
  Riebisch et al. propose using UML multiplicities because they represent both cardinalities, OR, XOR and AND relationships (group cardinalities) and, mandatory and optional features. The multiplicities are to be placed in the relation and not the features.

An interesting and original consideration in [72] is the semantic meaning of a feature, which is a child of two parent features each one with a different “level of optionality” for the child feature. Figure 3.1 illustrates the child feature, meal, and its two parent features, business and economy. Usually a customer flying in business class is provided with a meal, but this is not the case when flying economy class, though the service is frequently offered for an extra surcharge; therefore, the meal is mandatory for some (business) while its optional for others (economy).

![Feature Diagram Example]

**Figure 3.1:** Example illustrating Riebisch et al. feature diagram graph representation

- [Svahnberg et al. 82]
  Svahnberg et al. propose an extension to FeatuRSEB but the changes they add are about the syntax hence the underlying FeatuRSEB characteristics remain untouched. However, their novelty is the integration of “binding times” (when a feature can be selected), and “external features” (related target platform features).
3.2 Directed hyper-graph modelling of feature diagrams

Directed acyclic hyper-graphs may be of advantage to model a feature diagram by means of abstracting either its tree or its directed acyclic graph representation model as claimed by Laguna and Marqués in [53] [54].

The reasons argued by Laguna and Marqués for this novel approach is the substitution of many of the decomposing relationships (AND, XOR and OR) and types of features (single or grouped), used in many other approaches, such as the ones described in the previous subsection. Two major elements are proposed in [54]:

- **Features**
  Set of nodes of the hyper-graph representation.

- **Generic decomposition relationships**
  Set of hyper-arcs connecting the features (nodes).

The relationships, which are the most important semantic elements in a feature diagram because they associate the features between themselves, are modelled as forward hyper-arcs. In similar fashion as a tree model of a feature diagram, each decomposing relationship is modelled with a labelled f-arc, where the label is the multiplicity of the original decomposition.

Below figure 3.2 illustrates a feature diagram using Czarnecki et al. notation [21] [23] based on the feature model describing the features of a simple telecommunication system posted by Alexander Felfernig at SPLOT [60], and the equivalent hidden directed acyclic hyper-graph. Blue and red coloured arcs respectively depict exclude and require constraints that cannot be modelled, continue reading this section for further explanations about their meaning.

The corresponding transformation of the feature diagram depicted in 3.2 is a f-hyper-graph because all feature diagram decomposing relationships have one origin feature and one or more destination features (the illustration highlights the transformation of the constraints which will be explained below); therefore, each one will be transformed into an f-arc, and a hyper-graph whose hyper-arcs are f-arcs, is named f-hyper-graph (or simply f-graph).

Owing to the fact that a direct transformation of each decomposing relationship into a labelled f-arc is not enough, Laguna and Marqués also classify each labelled f-arc as follows, which considers all types that Schobbens et al. considered relevant in their comparison study [76]:

- **Mandatory relationship**
  A f-arc \( E = (T_E, H_E) \) such that \(|H(E)| = 1 \land label(E) = [1 \ldots 1] \)

- **Optional relationship**
  A f-arc \( E = (T_E, H_E) \) such that \(|H(E)| = 1 \land label(E) = [0 \ldots 1] \)

- **Alternative relationship** (XOR)
  A f-arc \( E = (T_E, H_E) \) such that \(|H(E)| = q / q > 1 \land label(E) = [1 \ldots 1] \)

- **“Addition” relationship** (OR)
  A f-arc \( E = (T_E, H_E) \) such that \(|H(E)| = q / q > 1 \land label(E) = [1 \ldots q] \)

---

5 Using the definition by Gallo et al. published in [30]

6 Later on, the validity of the proposed feature diagram’s hyper-graph model is justified using the equivalent transformation into a feature diagram’s directed acyclic graph model.

7 It is important to remark the acyclic constraint imposed on each hyper-arc and consequently on the hyper-graph to satisfy the validity of the definition of a hyper-arc by Gallo et al. in [30] which Nielsen and Pretolani proved weak in [65] unless the hyper-arc was acyclic.

8 Do not forget that a hyper-arc is a forward hyper-arc when the cardinality of its tail set is one (and the cardinality of its head set can be any; however, in a feature diagram it should be at least one because it would not have any semantic meaning an association of a parent feature with an empty set).
While a feature diagram would be very well represented as a f-graph using the previous transformations, there still are some semantics which can not be expressed in a feature diagram using features and decomposing relationships and additional constraints are required [23]. [Czarnecki and Kim] highlight the two best known: implies (Feature $A$ requires feature $B$) and excludes (Feature $A$ excludes feature $B$) constraints.

Laguna and Marquès also consider these two as f-arcs in order to fully describe the semantics of a feature diagram as a f-graph model:

- **Require constraint**
  A f-arc $E = (T_E, H_E)$ such that $|H(E)| = q / q ≥ 1 \land label(E) = [q \ldots q]$  
  The semantic meaning of a require hyper-arc is that the feature in the tail set imposes the constraint of selecting all features in the head set.  
  Laguna and Marquès point out that for this particular constraint transformation, it is feasible that cycles are created in the f-graph; therefore, it is utterly necessary that the feature diagram f-graph model has its aciclicity checked once all constraints have been transformed, which can be accomplished with the procedure described by Gallo et al. in [30].

- **Exclude constraint** (Mutex)
  A f-arc $E = (T_E, H_E)$ such that $T(E) = Root \land |H(E)| = q / q > 1 \land label(E) = [0 \ldots 1]$  
  $^9$Note that feature $B$ does not necessarily have to be one feature, it may very well be a set $S$ of features
The semantic meaning of a mutex hyper-arc is that it is not possible to select more than one of the features in the tail set at the same time.

The transformation of this particular constraint is peculiar in the sense of the correspondence between the feature diagram and the hyper-graph model. The constraint is formed between a set of features, but the modelling as a f-arc requires a source set and a tail set too, and the semantic meaning of a f-arc for this proposal is that when the tail set (which is a feature) is selected, the label (which is the cardinality) of the f-arc shall be met; but given the mutual exclusive relationship between all features involved, it is not possible to model one of them as the head set. For that reason, the head set of a f-arc modelling an exclude constraint is the root feature of the feature diagram because, by definition, it is the minimum feature set that any feature diagram can have.

Formally [53, 54], a feature diagram, $F = (\mathcal{N}, E, \nabla, \delta)$, is an acyclic forward hyper-graph (or simply acyclic f-graph) where:

- $\mathcal{N} = \{n_1, n_2, n_3, \ldots, n_n\}$ is the finite set of features (nodes)
- $E = \{E_1, E_2, E_3, \ldots, E_m\}$ / $E_i \subseteq \mathcal{N} \forall i = 1, \ldots, m \land |H(E_i)| = q_i$ is the set of decomposing f-arcs, each one with a head set cardinality of $q_i$
- $\nabla \in \mathcal{N}$ is the root feature of the feature diagram

The root feature is the single node in the hyper-graph, which is not part of any head set from any hyper-arc in the hyper-graph. Given the root set $\mathcal{R}$ of the hyper-graph, the following applies:

- $\mathcal{R} \subseteq \mathcal{N} \land \mathcal{R} = \{\nabla\}$
- $BS(\nabla) = \emptyset$
- $BS(n) \neq \emptyset$ / $\forall n \in (\mathcal{N} - \mathcal{R})$

- $\delta : E \rightarrow M / M \subset \mathcal{N} \times \mathcal{N}^*$ is the label function which assigns a multiplicity value $mv(E_i) = (min, max) \in M$ to each f-arc $E_i$ such that $min, max \in \mathbb{Z} \land min \geq 0, max > 0, min \leq max \land max \leq q_i$

There is a particular case when each feature in the feature diagram does not have more than one parent feature (naturally excluding the root feature); in this situation, the feature diagram is a hyper-tree where $|BS(n)| = 1$ / $\forall n \in (\mathcal{N} - \{\nabla\})$.

Figure 3.3: Telecommunications system f-graph transformation

Previous figure 3.3 illustrates the (full) f-graph transformation of the telecommunications system feature diagram depicted in figure 3.2 according to Laguna and Marqués hyper-arc transformation processes. Note that the hyper-edge representing the require constraint between Messaging and IMS and Juniper features,
could have been transformed in figure 3.2 into two separate hyper-edges of multiplicity \([1, 1]\) between \(\text{Messagging}\) and \(\text{IMs}\), and, \(\text{Messagging}\) and \(\text{Juniper}\); but for illustration purposes the transformation of the constraint modelling from the feature model is preserved\(^{10}\).

In addition to the preceding generic definition of a feature diagram modelled as an acyclic f-graph, \(\text{Laguna and Marqués}\) also identify two extensions with reference to constraints and type of features. Considering a feature diagram, \(F = (\mathcal{N}, \mathcal{E}, \mathcal{V}, \delta)\),

- a constrained feature diagram (CFD), \(C_F = (\mathcal{N}, \mathcal{E}', \mathcal{V}, \delta')\), is a feature diagram where:
  - \(\mathcal{E}' = \mathcal{E} \cup \mathcal{E}_r \cup \mathcal{E}_m\)
    * \(\mathcal{E}_r = \{r_1, r_2, r_3, \ldots, r_k\} \land r_i \in \mathcal{N} \forall i = 1, \ldots, k\) is the set of \textit{require} constraints (modelled as f-arcs, which, in general, \(|H(r_i)| \geq 1\))
    * \(\mathcal{E}_m = \{m_1, m_2, m_3, \ldots, m_l\} \land m_j \in \mathcal{N} \forall j = 1, \ldots, l\) is the set of \textit{exclude} constraints (modelled as f-arcs, which, in general, \(|H(m_j)| \geq 2 \land T(m_j) = \mathcal{V}\))
  - \(\delta' = \delta \cup \delta_r \cup \delta_m\)
    * \(\delta_r : \mathcal{E}_r \rightarrow \mathcal{M}\) is the label function which assigns a fixed multiplicity value \(mv(r_k) = (\min, \max) \in \mathcal{M}\) to each f-arc \(r_k\) such that \(\min = |H(r_k)| = \max\)
      When the \textit{require} constraint involves two nodes, \(mv(r_k) = (1, 1)\)
    * \(\delta_m : \mathcal{E}_m \rightarrow \mathcal{M}\) is the label function which assigns a fixed multiplicity value \(mv(m_l) = (\min, \max) \in \mathcal{M}\) to each f-arc \(m_l\) such that \(mv(m_l) = (0, 1)\)
- a typed feature diagram (TFD)\(^{11}\), \(T_F = (\mathcal{N}, \mathcal{E}, \mathcal{V}, \delta, \tau)\), is a feature diagram where:
  - \(\mathcal{P}\) is the set of the following types: \(\text{INTEGER, REAL, BOOLEAN, STRING, NONE}\)
    \(\text{NONE}\) type feature would be the default one
  - \(\tau : \mathcal{L} \rightarrow \mathcal{P}\) is the function which assigns a value to each leaf\(^{12}\) feature such that \(\mathcal{L} \subset \mathcal{N} \land |FS(l)| = 0 \forall l \in \mathcal{L}\)

\(\text{Laguna and Marqués}\) demonstrate how an acyclic forward hyper-graph model of a feature diagram is analogous to previous models of feature diagrams (for instance, those explained at the beginning of the current section). Hypothesising a transformation of the f-graph into a directed acyclic graph, where:

- each f-arc such that its cardinality’s tail set is one, \(|H(E)| = 1\), is to be transformed into an \textit{AND} or an \textit{OPTIONAL} decomposition (and the f-arc’s multiplicity is set to the feature)
- each f-arc such that its cardinality’s tail set is more than one, \(|H(E)| > 1\), is to be transformed into a grouped decomposition (and the f-arc’s multiplicity is set to the decomposition)

The transformed feature diagram is compatible with the generic semantic framework that \(\text{Schobbens et al.}\) defined in [76] to “define, study and compare” feature diagram languages, called Free Feature Diagram (FFD), useful to describe (or define) any feature diagram syntax regardless of the semantics, with reference to the criteria of expressiveness, embeddability, succinctness and redundancy.

Consequently, (acyclic forward) hyper-graph modelling of feature diagrams proposed by \(\text{Laguna and Marqués}\) is a satisfactory transformation, which has been proved to be equivalent to the many of the most important feature diagram representation proposals as a result of its proved compatibility (without loss of information) with them.

\(^{10}\)Moreover, expanding these kind of constraints will impact the performance of an algorithmic procedure because the size of the hypergraph will be increased \(\text{[83]}\)

\(^{11}\)\(\text{Laguna and Marqués}\) remark that they consider this extension as optional because there is not a general agreement among scientific community about typed features.

\(^{12}\)A leaf feature is any feature \(n\) such that its forward star set is empty (it is not included in any tail set of any hyper-arc in the hyper-graph); that is, \(FS(n) = \emptyset\)
Chapter 4

Verification of feature diagrams

“Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity”

Alan Turing

4.1 Graph traversal

Traversal is the process of visiting (examining, updating or performing an operation) each node (or edge) of the set of graph’s nodes (or edges), precisely once, in a systematic (algorithmic) fashion [93].

For instance, one would want to traverse a graph to find the answer for “which node is at the head of this edge?”, “which edges are outgoing from this node?” or “is there a path between these two nodes?” [73], where the first two questions require a single step traversal operation, and the latter question may require more than one step traversal operation.

Some classical graph traversal algorithms [1, 19] are Depth-First Search, Breadth-First Search and Single-Source Shortest Path, which are available in appendix D.

Among the algorithms listed in the appendix, there are two variations suitable for directed graphs (and obviously, acyclic too): Directed Depth-First Search (listing D.3) and Directed Single-Source Shortest Path (listing D.5).

The so popular Dijkstra’s algorithm (listing D.4), which can even be used in weighted graphs, is already suitable for directed graphs although it is not viable when the weights are negative (in case of negative weights, Bellman-Ford’s algorithm is of use).

These algorithms are not suitable for hyper-graph traversal, for the most part as a consequence of the size of the head and tail set of a hyper-edge (or hyper-arc), but they constitute the foundations of the algorithms which will be shown in the next section.

The strategy is the same, find a path connecting two nodes according to some constraints. Dijsktra’s constraint is finding the shortest path (which will be the path will the smallest weight) connecting two nodes [28], but Depth-First Search or Breadth-First Search are used to find a node, and this search is done according to some constraints (the former, DFS, uses a policy of searching the deepest branches first, while the latter, BFS, uses a policy of searching level by level).

4.1.1 Applications

The strength of graph traversal algorithms relies on the applications where they are more suitable than other algorithms by reason of the traversal procedure techniques. Some of those applications are listed for illustration purposes as follows:

• Edge classification
DFS for directed graphs can classify the edges into its strongly connected components [1].
As a result of running DFS algorithm on a (directed) graph, its edges can be categorised into tree edges, back edges, forward edges and cross edges [19].

- **Acyclicity**
  DFS can be used to verify that a graph has no cycles (there is a cycle if a back edge relative to its DFS forest exists) [1].

- **Ordering**
  DFS can generate a linear order of the vertices so that is consistent with the partial order defined by its edges [1, 19].

- **Longest path**
  According to Atallah and Blanton, computing the longest path of any graph is NP hard; however, if the graph is directed and acyclic, the computation can be done in linear time [1, 19].
4.2 Hyper-graph traversal

Previous section reflected on how the problem of traversing a graph is not trivial nor unchallenging, and it also included the pseudocode of some classical algorithms, which the reader can find in appendix D.

Nonetheless, when traversing a hyper-graph, the approach differs slightly because the structure is more abstract but also more powerful and robust. The hyper-graph traversal algorithms which will be covered in this section are the basis of the algorithms on which the verification algorithms for feature diagrams are constructed.

These traversal algorithms are also proposed by Gallo et al. in [30], so the definition of a hyper-graph and the descriptions about hyper-arcs and hyper-paths described in section 2.3.3 shall be considered for the following algorithms.

4.2.1 Visit

Listing 4.1 illustrates the traversal of a hyper-graph $H$ from an origin node $r$ to find out all the nodes connected to it. The algorithm, which runs in $O(\text{size}(H))$ time, finds all the nodes connected to this node, and returns these nodes along with the paths connecting these nodes to the node $r$.

These paths have a structure of a tree rooted at node $r$. In order to achieve this tree, after an initialisation phase (the first two for loops), which runs in $O(n + m) / n = |\mathcal{V}| \land m = |\mathcal{E}|$ time, there are two (predecessor) functions of interest in the main traversal loop:

- $P_v(i)$: returns the hyper-arc $E_j \in BS(i)$ such that it precedes the node $i$ in the path
- $P_e(E_j)$: returns the node $i \in T(E_j)$ such that it precedes the hyper-arc $E_j$ in the path

```
1 procedure visit(r, $\mathcal{H} = (\mathcal{V}, \mathcal{E})$)
3     for each $i \in \mathcal{V}$ do
4         $P_v[i] := 0$;
5     end for
7     for each $E_j \in \mathcal{E}$ do
8         $P_e[E_j] := 0$;
9     end for
11     $P_v[r] := \text{nil}$;
12     $Q := \{r\}$;
14     repeat
15         $i := \text{select}(Q)$; // Select an element in $Q$ (e.g. LIFO manner)
16         $Q := Q - \{i\}$;
17         for each $E_j \in FS(i)$ such that $P_e[E_j] = 0$ do
18             $P_v[E_j] := i$;
19             for each $h \in H(E_j)$ such that $P_v[h] = 0$ do
20                 $P_v[h] := E_j$;
21                 $Q := Q \cup \{h\}$;
22             end for
23         end for
24     until $Q = \emptyset$;
26 end procedure
```

Listing 4.1: Hyper-graph visit procedure [30]
4.2.2 Backward-visit

Algorithm shown in listing 4.1 can be improved when some restrictions are considered. One of these restrictions is the connection between two (distinct) nodes, for instance, a B-connection. The hyper-graph traversal algorithm illustrated in the pseudocode depicted in listing 4.2, which runs in $O(|H|)$ time, finds all the nodes B-connected to a node $r$, and returns these nodes along with the paths B-connecting these nodes to the node $r$.

In the same way as the previous algorithm, the paths have a structure of a B-tree\(^1\) rooted at node $r$. Even though the same functions of interested used in the previous algorithm are also employed for this one, Gallo et al. remark that the only the first one, $P_v(i)$, is required; but the second one, $P_e(E_j)$, is also kept because it is essential for one new function:

- $v(h)$: returns the cardinality of the path $P_{rh}$ in the tree defined by $P_e$ and $P_v$

```plaintext
1 procedure b-visit (r, $H = (V', E)$)
3 for each $i \in V'$ do
4 $P_v[i] := 0$;
5 $v[i] := \text{inf}$;
6 end for
8 for each $E_j \in E$ do
9 $P_v[E_j] := 0$;
10 $k_j := 0$;
11 end for
13 $P_v[r] := \text{nil}$;
14 $Q := \{r\}$;
15 $v[r] := 0$;
17 repeat
18 $i := \text{select}(Q)$; // Select an element in $Q$ (e.g. LIFO manner)
19 $Q := Q - \{i\}$;
20 for each $E_j \in FS(i)$ do
21 $k_j := k_j + 1$;
22 if $k_j = |T(E_j)|$ then
23 $P_v[E_j] = i$;
24 for each $h \in H(E_j)$ such that $P_v[h] = 0$ do
25 $P_v[h] := E_j$;
26 $Q := Q \cup \{h\}$;
27 $v[h] := v[P_v[E_j]] + 1$;
28 end for
29 end if
30 end for
31 until $Q = \emptyset$
33 end procedure
```

Listing 4.2: Hyper-graph b-visit procedure [30]

It is possible to define\(^2\) a F-visit procedure so that it returns a set of F-paths, each one including all the nodes F-connected to node $r$, which is also the terminal node of the F-path.

1B-tree stands for backward-tree, a tree whose paths from any node to the node $r$ are b-paths. An equivalent definition would be a set of b-paths containing all nodes b-connected to the node $r$
2Given that its definition and pseudocode are practically the same as those for the B-visit, these are left as an exercise for the reader
Similarly a BF-visit procedure could be defined, but Gallo et al. note that it will not be easy unless the hyper-graph is a B-graph (F-graph or BF-graph).
4.3 Verification and configuration

As described in section 2.2.2, a feature model describes (graphically) the configuration alternatives for a product line or a system of families. Among all these possible configurations, one of the stakeholders involved in the software development process accomplishes a particular configuration by means of a manual process where desired features are selected taking the feature diagram and the textual or graphical constraints into account when carrying out the selection as well.

The resulting configuration, which is another feature diagram, is always a subset of the initial feature diagram (this process was explained in section 2.2.2 too, but it is of importance to remark that when configuring, it is not allowed to add new features).

There are a few rules that apply to a feature diagram configuration, which Laguna and Marqués have identified in [53]:

- **Core selection**
  Since the root feature of any feature diagram is the smallest prospective configuration, any mandatory child feature of the root feature (and subsequently, any other mandatory child feature connected indirectly to the root feature through mandatory child features) also belongs to this ‘smallest’ configuration.

- **Selection by ‘inheritance’**
  Any mandatory child feature of a selected feature should also be selected in the configuration.

- **Selection by ‘parenting’**
  A non-mandatory child feature of a feature can only be selected (or included in the configuration) if it has at least one parent which is selected.

- **Decomposing selection**
  When a feature is selected, the number of its child features which are to be selected should not more than \( n \) but no less than \( m \), for a cardinality of \([m \ldots n]\)

- **Selection by require constraint**
  Any feature which is required as a result of a selected feature for which there is a require constraint such that demands it should also be selected in the configuration.

- **Selection by mutex constraint**
  Any selected feature which is involved in a mutex constraint restricts the selection in the configuration of all other features taking part in the same mutex constraint.

Moreover, Laguna and Marqués also reformulated these rules as two properties that the subset of features of the feature diagram (the configuration), excluding the root feature, shall abide by:

- **Given an structure feature from the configuration, one or more of its parents should be present in the configuration as well.**
  This property satisfies the third rule.

- **Given a hyper-arc, \( e \), with a multiplicity value, \( mv = [min \ldots max] \), whose tail (feature) is selected, no less than \( min \) and no more than \( max \) features of the hyper-arc’s head (child features) should also be present in the configuration.**
  This property satisfies all other rules which previous property does not.

When \(|H(e)| = 1\) (children’s cardinality set is one):

---

3 ‘Core’ should be read as in minimal, the least possible
4 Considering a valid configuration, that is, a feature diagram modelled as a forward hyper-graph with acyclic hyper-arcs
5 The tail of a hyper-arc which is not the transformation of a require or mutex constraint
4.3. Verification and configuration

– if \( \text{min} = 1 = \text{max} \), the feature is mandatory, and should present if the parent is (first and second rules), or it is a require constraint and the child should also be present (fifth rule).

– if \( \text{min} = 0 \land \text{max} = 1 \), the feature is optional

When \(|H(e)| > 1\) (children’s cardinality set is more than one):

– if \( \text{min} = 1 = \text{max} \), it is a XOR feature group, and only one of the children should be present at most if the parent is present.

– if \( \text{min} = 0 \land \text{max} = 1 \), it is an optional feature group, and child features can be present or not as long as its parent is present, or it is a mutex constraint and at most one of the child features can be present (sixth rule).

– if \( 1 \leq \text{min} < \text{max} \leq |H(e)| \), it is a OR feature group, and no more than \( \text{max} \) and no less than \( \text{min} \) child features can be present if the parent feature is present.

In general, if \( 0 \leq \text{min} < \text{max} \leq |H(e)| \), no more than \( \text{max} \) and no less than \( \text{min} \) child features can be present if the parent feature is present.

Finally, Laguna and Marqués consider in [53] that a feature diagram is valid if at least a configuration can be derived from it, and if each feature is present in at least one configuration (that is, there are no dead features).

4.3.1 Configuration process

Czarnecki et al. define configuration in [22] as the process of specifying a family member, whereas the specification is fulfilled by a stakeholder (although, it is usually an application engineer) who selects the desired features from the feature model taking into account the variability constraints defined by the model as well as other constraints which could not be modelled as a feature diagram. For instance, deselecting an optional feature so that it will not be in the configuration, or choosing two features among a set of OR alternative features.

However, despite this process of configuring a feature model is undertaken by a trained stakeholder, it is frequent that some decisions are not taken, that is, variability points exists in the final configuration. These variability points are not acceptable if a product has to be made, but neither is that there are contradictory decisions such as selecting more than one feature among a XOR alternative feature group.

For these reasons, any configuration must be verified and completed if the selection did not define a unique configuration (that is, one or more partial configurations). An obvious, but highly expensive in terms of computing resources, methodology is finding all complete configurations given the partial configuration(s), while another procedure, which Laguna and Marqués agree with, is performing a guided refinement process where the application engineer is involved.

This specification procedure in stages, where each step eliminates one or more variability points, because the application engineer makes some configuration choices, is known as staged configuration, and was proposed by Czarnecki et al. in [21]. Given a feature model, each stage produces a specialisation of the feature model, where the set of products described by in the specialisation is a subset of the set of products described in the feature model which was specialised.

Laguna and Marqués propose a two staged configuration process, where each stage satisfies the corresponding property explained earlier:

- **Generation of partial configuration(s) (First stage)**

  Given a set of selected features, \( \mathcal{U} \subseteq \mathcal{N} \), this first stage should find a (valid) path between each feature,

---

6 At this point, the reader must have an idea of the trivial cases; those are, a configuration with one feature, the root feature, sole feature required for a feature model to be valid, and a configuration which satisfies the first rule of the six previous described rules.

7 They assume that feature model to be specialised is well formed and does not have any inconsistency. For instance, since the underlying data structure is a forward hyper-graph, there should not be any cycle, although this can be easily verified [30, 34].
Considering a feature diagram, \( \mathcal{F} = (\mathcal{N}, \mathcal{E}, \nabla, \delta) \) and a set \( U \subseteq \mathcal{N} \) of (manually) selected features, Laguna and Marqués define a valid configuration in [53]. \( \mathcal{G} = (\mathcal{N}_G, \mathcal{E}_G, \nabla) \) / \( U \subseteq \mathcal{N}_G \), as a sub-hyper-graph of a constrained feature diagram (CFD), \( \mathcal{C}_G = (\mathcal{N}, \mathcal{E}', \nabla', \delta') \), where:

- \( \nabla \in \mathcal{N}_G \) (Root is always selected)
- \( \mathcal{N}_G \subseteq \mathcal{N} \)
- \( \mathcal{E}_G = \{ e_G \mid \exists e \in \mathcal{E}' \land T(e_G) = T(e) \land H(e_G) \subseteq H(e) \} \)
- \( \exists e \in \mathcal{E}' \land \text{type}(e) \neq \text{constraint} \land n \in H(e) \land T(e) \in \mathcal{N}_G \forall n \in \mathcal{N}_G / n \neq \nabla \) (Each selected feature has at least one parent feature selected too)
- Given \( H'(e) \) as the head of any hyper-arc, \( e \in \mathcal{E}_G \), \( H(e) \) in configuration \( \mathcal{G} \) is a subset of the original \( H(e) \) in feature diagram \( \mathcal{F} \) such that:
  - \( \forall e \in \mathcal{E}_G, |H'(e)| > 0 \land H'(e) \subseteq H(e) \)
  - \( \forall n \in \mathcal{N}_G \land e \in \mathcal{E}_G \land T(e) = n \Rightarrow \min(e) \leq |H'(e)| \leq \max(e) \)

That is, for each selected feature in the configuration, between \( \min \) and \( \max \) child features of the original (associated) hyper-arc are present.

First stage: generation of partial configuration(s)

The first stage is a customised implementation of the b-visit hyper-graph traversal algorithm (see listing 4.2 in section [4.2] from Gallo et al.). Its complexity is the same as b-visit traversal algorithm, that is, \( O(\text{size}(\mathcal{H})) \), although it is increased for every feature with more than one structural feature parent (hence, the complexity is multiplied by the number of incoming hyper-arcs excluding the root feature concept).

Given a constrained feature diagram (CFD), \( \mathcal{C}_G = (\mathcal{N}, \mathcal{E}', \nabla, \delta') \), a set \( U \subseteq \mathcal{N} \) of (manually) selected features and an initial configuration, \( \mathcal{G}_0 = (\mathcal{N}_0, \mathcal{E}_0) / \mathcal{N}_0 = \{ \nabla \} \land \mathcal{E}_0 = \emptyset \) (the initial configuration only has the root node selected and no hyper-arcs), the procedure, whose pseudocode is illustrated in listing 4.3, works as follows:

---

This could be categorised as an economical factor.

48
4.3. VERIFICATION AND CONFIGURATION

- A feature \( u \in U \) is selected, and removed from \( U \).
- For each structural hyper-arc, \( e \in E' \), whose head contains the selected feature \( u \) and the maximum cardinality of the hyper-arc has not been met, the algorithm selects the tail, \( T(e) \), of the hyper-arc if it was not already, adds it to the \( U \) set and a recursive call is executed (with the updated set \( U \) and the partial configuration \( G' \)).
- If \( U = \emptyset \), the algorithm is finished; otherwise it continues with the first step.

The implemented version of the pseudocode includes some novelties of interest which improve the overall process of finding many partial configurations while discarding those which are not valid:

- Every time a feature is selected, it is also checked for mandatory descendant features (which will consider require hyper-arcs).
- Any feature which was not part of the initial selection and results selected due to the algorithm, is checked against the feature diagram to verify the compatibility, and if there is one feature which makes the selection incompatible, it is automatically discarded.
- If a feature has more than one parent feature, the algorithms forks as many times as parent features, creating a new partial configuration for each of those parent features.
  This is the main reason for the algorithm to return a set of partial configurations, but if the feature model does not have any feature with more than one parent feature, then it will return one partial configuration.

Second stage: completing partial configuration(s)

The second stage is a customised implementation of the f-visit hyper-graph traversal algorithm (see listing 4.2 in section 4.2 from Gallo et al.). Its complexity should be similar to f-visit traversal algorithm, but as a consequence of internal calls to first stage algorithm for features added via require extra constraints, the original complexity of f-visit, \( O(\text{size}(H)) \), becomes a lower bound for this implementation instead.

Given a set of partial configurations, \( G'_i = (\{ N_0, N_1, N_2, \ldots, N_n \}, \{ E_0, E_1, E_2, \ldots, E_n \}) / \forall N_i \subseteq N_i \land \forall E_i \subseteq E \), the procedure, whose pseudocode is illustrated in listing 4.4, works as follows:

- A list of structural hyper-arcs \( W \) whose tail feature is selected and their minimum cardinality is not met, is generated.
- For each hyper-arc, \( e \in E_i \), the algorithm selects as many features as required to meet the minimum cardinality, and a recursive call is executed.
- If \( W = \emptyset \), the algorithm is finished; otherwise it continues with the first step.

The implemented version of the pseudocode includes some novelties of interest which improve the overall process of completing each \( G_i \) partial configuration while discarding those which are not valid:

- Every time a feature is selected, it is checked for mandatory descendant features (which will consider require hyper-arcs).

In this case, it is known that mandatory descendant features of the selected feature will be connected to this feature, either directly (because they are its children) or indirectly. But those features selected via require constraints might not have any parent feature selected; therefore, the first stage algorithm should be executed again, where these selected features via require hyper-arcs are the \( U \) selection set because a path to the root must be found. And once finished, the second stage algorithm should, again be executed because they will have generated partial configurations.

\[^9\text{Work in progress, noted in section 5.1 to elude these calls and simplify the complexity of the second stage algorithm, is currently underway.}\]

49
• Any feature which was not part of the initial selection and results selected due to the algorithm, is checked against the feature diagram to verify the compatibility, and if there is one feature which makes the selection incompatible, it is automatically discarded.

The policy which decides how many features have to be selected (excluding meeting the minimum of every hyper-arc which has to be completed anyways), and which ones among those belonging to the head of the hyper-arc is yet to be implemented. The prototype implementation, considered meeting the minimum cardinality and in no particular order, that is, the order was set by the strategy used when reading the feature model from the XML file.

However, this is not a trivial question which can be underestimated or ignored. When implementing a product line, a set of policies for this default behaviours must be specified.

Verification and configuration algorithms pseudocode

• First stage: Generating (partial) configuration(s)

Listing 4.3: Feature diagram (partial) configuration procedure [53]
• **Second stage**: Completing (partial) configuration(s)

```java
procedure complete(G_i)
    for each e ∈ E and getMark(tail(e)) do
        W := W ∪ {e};
    end for
    while W ≠ ∅ do
        // Select and remove node w ∈ W
        w := first(W);
        W := W − {w};
        // Try non-selected features of hyper-arc head up to min cardinality
        for each f ∈ head(w) do
            // Select feature f and check for no conflicts
            if used(w) < min(w) and not getMark(f, G_i) then
                mark(f, G_i);
                used(w) := used(w) + 1;
                if compatible(G_i) then
                    // Find partial configurations for require constraints
                    if count(getRequire(G_i)) > 0 then
                        G_i := configure(getRequire(G_i), G_i);
                    else
                        G_i := G_i;
                    end if
                    // Complete derived partial configurations or current
                    for each g ∈ G_i do
                        G_i' := complete(g);
                        if compatible(G_i') then
                            G_i := G_i ∪ {G_i'};
                        else
                            // Configuration not valid
                            G_i := void;
                        end if
                    end for
                else
                    // Configuration not valid
                    G_i := void;
                end if
            else
                // Configuration not valid
                G_i := void;
            end if
        end if
    end while
    if W = ∅ then
        // Configuration is completed
    end if
end procedure
```

**Listing 4.4**: Feature diagram completing (partial) configuration procedure [53]
4.4 Experimental evaluation

Any scientific hypothesis is meaningless without proof. Sometimes that proof is one or more theoretical demonstrations but some other times, the proof comes in the form of results such as those obtained from running experiments on the practical implementation of the scientific hypothesis, which can also supplement a theoretical demonstration.

This section will show the results of one practical implementation of the hyper-graph traversal algorithms customised for feature diagram configuration (as a two-stage process configuration), shown in the previous section, 4.3.

These results are merely the culmination of the background theory, methods and processes which have been documented throughout previous sections and chapters of this thesis, which in turn are the theoretical demonstrations for the scientific hypothesis of feature diagram modelling by means of a forward (acyclic) hyper-graph proposed by Laguna and Marqués in [54].

The implementation of the algorithms was accomplished with Java™, and the version used was the latest available at the moment of the experimentation, 1.6.0_17.

The reason for using this imperative programming language instead of a language more suited for the application of functions and recursion seeing that the algorithms are highly recursive and dependant on themselves (in all probability because of the strong mathematical background behind the traversal algorithms for hyper-graphs) such as any functional programming language is because nowadays neither the programming paradigm nor the programming language itself do not seem to have much influence on the speed nor the memory footprint [29].

Consequently, the decision revolved around using a programming language which was easy to learn and use, with a good Application Programming Interface (API) library to minimise memory leaks and non-efficient implementations of standard operations, which was close to C# and possibly object-oriented.

4.4.1 Technical details

All experiments were executed with the following appliances and computer applications:

- **Hardware equipment**
  - Processor: Intel™ Core 2 Duo 2.16 Gigahertz
  - Memory: 2048 Megabytes

- **Computer software**
  - Operating system: Mac™ OS X 10.6.3 (10D573)
  - Runtime environment: Java™ 6 Standard Edition 6 (1.6.0_17-b04-248-10M3025)
  - Runtime mode: 64 bits
  - Libraries: Java 6 Standard API, Simple XML Feature Model (SXFM) [60]

- **Other considerations**
  - Command line: java -Xmx512m FDVerVal model.xml
  - Profiler: YourKit Java Profiler 9.0.0 EAP (5052)

---

10When comparing with C GNU / gcc and C++ GNU / g++, which is an industry standard practice
11This was not a strict requirement because the author had freedom to choose the language but it was a recommendation given that the implementation would later be imported into the tools being developed within the research group
12Runtime environment bundled by Apple Inc. with its latest operating system update, whose upstream provider is Sun Microsystems Inc.
Sampling test feature models

Early stages of the implementation phase were debugged using the feature model depicted in figure 4.1 while later debugging phases were debugged using the feature model illustrated in figure 4.2.

The former feature model is a ‘real’ problem for software product line evaluation proposed by López-Herrejon and Batory in [56]. This model is intriguing because it has many require constraints, which are the ‘problematic’ ones for the second stage of the configuration algorithms. The latter feature model is a ‘customised’ problem which was built ad-hoc while implementing and debugging the configuration algorithms for both stages, and the aim for this model was testing the non-trivial case of a child feature having more than one feature parent.

Figure 4.1: Feature diagram of a graph product line [56]

Figure 4.2: Debugging feature diagram sample

Listing 4.5 illustrates the previous graph product line depicted in figure 4.1 coded in the format proposed by
Mendonça in [60], Simple XML Feature Model (SXFM), employed in all all tests since the sample test were acquired from SPLOT. Note that constraints are formulated as conjunctive normal forms (see appendix C).

Listing 4.5: XML model of graph product line feature diagram [56]
4.4. Experimental evaluation

Data structure-design

Illustration in figure 4.3 depicts the class diagram, modelled in Unified Modelling Language 2.2 (UML2) [68], of the hyper-graph and feature model hierarchy used as core data structures for the representation of a feature diagram as a forward (acyclic) hyper-graph.

The underlying hyper-graph data structure was extracted from “Oryx Editor” [71], a web-based Business Process Modelling Notation (BPMN) editor, leaded by Polyvyanyy. The hyper-graph data structure implementation used for this project was very appropriate for this thesis because it matched the definition by [30] of hyper-graph. Feature model data structure was built on top of this implementation, to take advantage of the internals and some functions; although some customisations were required in the underlying hyper-graph data structure as well.

Figure 4.3: Class diagram of feature diagram and hyper-graph representations
4.4.2 Results

The results that follow were carried out on the computer appliances described in section 4.4.1 using two different sets of test feature models.

The first set was composed of feature diagrams which were downloaded from the SPLOT project [60]. At first no considerations about the models were taken into account, that is, all models were supposed to be tested. However, after careful examination of the models, some of them had to be discarded, because they did not have extra constraints (they were, what Mendonça described in [59], taxonomies), the depth of the tree was just one or two levels or the number of features from the feature diagram involved in the extra constraints was too low (less than four).

These reasons and the varying number of features for each feature diagram caused that the experimentations on real feature models were divided into three groups according to the number of features (size) of the feature diagram: small (models with more than 10 features but less than 25), medium (models with at least 25 features but less than 50) and big (models with more than 50 features). And each group would not have more than eight models (usually it was two or three models per decade when the range was limited to a range).

The second set was also assembled with feature diagrams downloaded from SPLOT, but these models were for benchmarking purposes, as the number of features was extremely high (up to 10000 features), and the complexity of the extra constraints (up to 1000 constraints) was also extreme because, as opposed to the first set where the extra constraints were made of no more than two clauses (in some isolated cases there was one or two extra constraints of three clauses), in these models, all extra constraints were made of three clauses, and the approach taken to transform these into a feature diagram (see appendix C), is a conversion to a require constraint in order to emulate the behaviour in a hyper-graph.

These type of constraints are the ones which increase the traversal time because they are hyper-arcs pointing to any feature in the feature diagram and specially, in the second stage, each time a require hyper-arc has to be completed, a call to the first stage algorithm has to be made, and once finished, continue with the completing process.

Another important remark is the consistency of the models of all sets. SPLOT project aims at providing consistent models (excluding some benchmark models which are clearly marked as non-consistent), and the algorithms developed and tested in this thesis assume the same, that is, there should be at least one configuration if the model is to be traversed with the algorithms.

One of the models of the first set, “Electronic shopping”, was also tested with some customisations. These changes were aimed at testing the performance of the first and second stage configuration algorithms when one or more features have more than one feature parent (whose semantical meaning creates two alternative partial configurations for the engineer, designer or user to decide which parent feature is the desired one). These customisations where randomly introduced but making sure that the consistency of the model would not be affected.

The following is a description of the terms used in the header of each table to identify a name, value or measure. Note that average measurement times are in milliseconds for the first set (real feature models) but in seconds for the second set (benchmark feature models).

- **Feature Model Size (FM₅)**
  Number of variables (features) in a feature diagram.

- **Extra constraints representativeness (ECR) [59]**
  Ratio of the number of variables in the extra constraints (repeated variables counted once) of a feature model to the number of variables (features) in the feature diagram.

---

13SPLOT project features a tool to generate random feature diagrams of any size and giving the possibility of customising parameters such as the percentage of features involved in extra constraints (CNF), or the maximum number of features per grouped feature,...
4.4. EXPERIMENTAL EVALUATION

- **Relative configuration size (RCS)**
  Ratio of the number of variables in the selection of a configuration to the maximum number of variables (features) which can be selected in the feature diagram.

- **Zero, first or second staged-configuration time** ($S_i / i = \{0, 1, 2\}$)
  - *Stage zero*: Average measurement time of the process of adding any mandatory descendant feature starting at the root feature and verifying the selection of a configuration.
  - *First stage*: Average measurement time of the first stage traversal algorithm (using the verified selection of stage zero).
  - *Second stage*: Average measurement time of the second stage traversal algorithm (using the partial configurations obtained in the first stage).

- **Configuration time** ($S_{[k−l]} / k, l = \{0, 1, 2\} \land k < l$)
  Sum of zero, first and/or second staged configuration (average) measurement times plus an overhead time of extra operations between each consecutive pair of staged configurations (depicted with the star symbol).

- **First or second staged-configuration size** ($C_j / j = \{1, 2\}$)
  Average ratio of the number of valid configurations obtained in first, $C_1$, or second, $C_2$, staged-configuration phase.

Table 4.1 illustrates the measured times when no feature has been selected ($RCS = 0\%$), that is, the core selection explained at the beginning of section 4.3 and its completion in order to have a valid configuration.

<table>
<thead>
<tr>
<th>Feature Model</th>
<th>FM$_S$</th>
<th>ECR</th>
<th>S$_0$</th>
<th>S$_1$</th>
<th>S$_2$</th>
<th>S$_{[1−2]}^*$</th>
<th>C$_1$</th>
<th>C$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collaborative web [5]</td>
<td>14</td>
<td>21.4 %</td>
<td>4.49</td>
<td>3.66</td>
<td>40.86</td>
<td>45.51</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Search engine [59]</td>
<td>14</td>
<td>28.6 %</td>
<td>3.17</td>
<td>6.3</td>
<td>3.05</td>
<td>9.86</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Monitor engine [9]</td>
<td>17</td>
<td>11.8 %</td>
<td>5.76</td>
<td>4.63</td>
<td>33.3</td>
<td>39.34</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Mobile phone [77]</td>
<td>20</td>
<td>10.0 %</td>
<td>33.16</td>
<td>22.42</td>
<td>40.01</td>
<td>67.64</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Simple graphs [56]</td>
<td>20</td>
<td>60.0 %</td>
<td>32.23</td>
<td>31.23</td>
<td>15.29</td>
<td>47.03</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Weather station [60]</td>
<td>23</td>
<td>17.4 %</td>
<td>2.4</td>
<td>7.62</td>
<td>26.54</td>
<td>34.62</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Keyword in context IS [81]</td>
<td>25</td>
<td>16.0 %</td>
<td>5.25</td>
<td>9.93</td>
<td>16.59</td>
<td>26.89</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Digital video system [80]</td>
<td>26</td>
<td>23.1 %</td>
<td>10.49</td>
<td>10.34</td>
<td>11.4</td>
<td>23.4</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Reference management [60]</td>
<td>31</td>
<td>45.2 %</td>
<td>2.48</td>
<td>2.18</td>
<td>8.15</td>
<td>10.58</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Graph editor [57]</td>
<td>32</td>
<td>23.3 %</td>
<td>27.92</td>
<td>4.47</td>
<td>51.02</td>
<td>56.31</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Web portal [61]</td>
<td>43</td>
<td>25.6 %</td>
<td>3.03</td>
<td>13.52</td>
<td>7.19</td>
<td>20.9</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Document generation [86]</td>
<td>44</td>
<td>29.5 %</td>
<td>22.55</td>
<td>28.31</td>
<td>154.14</td>
<td>187.88</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Laptop computer [60]</td>
<td>46</td>
<td>80.4 %</td>
<td>19.67</td>
<td>19.44</td>
<td>217.03</td>
<td>236.97</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Arcade game [60]</td>
<td>61</td>
<td>55.7 %</td>
<td>36.53</td>
<td>35.19</td>
<td>215.72</td>
<td>253.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Home Integration System [42]</td>
<td>67</td>
<td>11.9 %</td>
<td>4.05</td>
<td>3.99</td>
<td>38.69</td>
<td>63.82</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Ecological car [60]</td>
<td>94</td>
<td>4.3 %</td>
<td>10.65</td>
<td>10.2</td>
<td>190.47</td>
<td>202.2</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Electronic shopping [55]</td>
<td>287</td>
<td>11.8 %</td>
<td>64.72</td>
<td>51.19</td>
<td>344.21</td>
<td>400.25</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Table 4.1**: Time measurements for core configuration on all test feature models
Tests on small-sized feature models

Sizes of feature models for this set of tests ought to be less than 25 features, some models were discarded because of an ECR = 0% as well, and some others because they were extremely simple.

The number of random generated configurations according to the RCS values shown in table 4.2 that is, 10 %, 15 %, 20 % and 25 %, were, 20, 15, 10 and 5 respectively.

<table>
<thead>
<tr>
<th>FEATURE MODEL</th>
<th>FM₅</th>
<th>ECR</th>
<th>RCS</th>
<th>S₀</th>
<th>S₁</th>
<th>S₂</th>
<th>S₁₋₂</th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collaborative web</td>
<td>14</td>
<td>21.4%</td>
<td>10 %</td>
<td>5.21</td>
<td>2.14</td>
<td>21.34</td>
<td>24.27</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>2.81</td>
<td>0.14</td>
<td>18.93</td>
<td>19.18</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>2.44</td>
<td>1.79</td>
<td>6.91</td>
<td>8.82</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>1.71</td>
<td>0.53</td>
<td>1.37</td>
<td>2.01</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Search engine</td>
<td>14</td>
<td>28.6%</td>
<td>10 %</td>
<td>0.6</td>
<td>0.18</td>
<td>1.82</td>
<td>2.1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>1.07</td>
<td>0.18</td>
<td>0.55</td>
<td>0.81</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>0.77</td>
<td>0.61</td>
<td>1.12</td>
<td>1.83</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>1.05</td>
<td>1.94</td>
<td>1.87</td>
<td>4.59</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Monitor engine</td>
<td>17</td>
<td>11.8%</td>
<td>10 %</td>
<td>1.01</td>
<td>0.17</td>
<td>1.37</td>
<td>1.66</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>1.91</td>
<td>0.19</td>
<td>3.15</td>
<td>6.77</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>7.12</td>
<td>0.19</td>
<td>1.49</td>
<td>1.83</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>4.01</td>
<td>0.23</td>
<td>0.75</td>
<td>1.17</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Mobile phone</td>
<td>20</td>
<td>10.0%</td>
<td>10 %</td>
<td>8.74</td>
<td>0.49</td>
<td>6.27</td>
<td>6.93</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>9.79</td>
<td>0.38</td>
<td>6.25</td>
<td>6.85</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>1.7</td>
<td>0.54</td>
<td>2.58</td>
<td>3.27</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>10.95</td>
<td>1.33</td>
<td>1.56</td>
<td>3.05</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Simple graphs</td>
<td>20</td>
<td>60.0%</td>
<td>10 %</td>
<td>4.32</td>
<td>0.42</td>
<td>3.78</td>
<td>4.78</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>3.52</td>
<td>1.31</td>
<td>2.39</td>
<td>3.83</td>
<td>0.87</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>6.54</td>
<td>1.38</td>
<td>1.59</td>
<td>3.09</td>
<td>0.56</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>2.34</td>
<td>3.68</td>
<td>4.07</td>
<td>7.88</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>Weather station</td>
<td>23</td>
<td>17.4%</td>
<td>10 %</td>
<td>3.07</td>
<td>0.18</td>
<td>7.06</td>
<td>7.36</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>3.49</td>
<td>0.74</td>
<td>3.85</td>
<td>5.92</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>2.56</td>
<td>0.35</td>
<td>4.36</td>
<td>4.93</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>5.59</td>
<td>2.96</td>
<td>9.68</td>
<td>7.8</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 4.2: Time measurements for staged configuration on small-sized test feature models

Analysis

Given the small size of these test feature models, there is not a distinguishable distinction between average times (excluding the “Collaborative web” feature model, which the reader can find an explanation for in the next set of
4.4. Experimental Evaluation

tests); however, the average ratio of valid configurations is visible in these set of tests and of interest.

Most of the tested models will show a $C_1$ and $C_2$ values of 1.0 or less. And the reason for this upper boundary is that most of these models are designed so that a feature does not have more than one parent feature, which is a case of particular interest for the evaluation of the algorithms presented in section 4.3.1, although it is rare that it happens but it may be required for some domains. Consequently, if all features in a feature model have no more than one parent feature, the partial configurations which the first stage configuration algorithm will return are at most one configuration.

Since the second stage configuration algorithm is fed from the results of the first stage algorithm, $C_2$ will not be greater than $C_1$ unless there is one or more features with more than one parent features, and one or more are selected during completion stage (check last set of tests for an example of multiple parent features).

“Simple graphs” feature model has a low $C_1$ and $C_2$ for $RCS = 15\%, 20\%, 25\%$ because most of the extra constraints (note that ECR is also quite high) are of require type which will result in many non-valid configurations, specially when selecting a high number of features for a configuration.
Tests on medium-sized feature models

As previously said, the sizes of feature models for this set of tests was required to be at least 25 and no more than 50 features, some models were discarded because of an $ECR = 0\%$, and others because the a low tree depth.

The number of random generated configurations according to the RCS values shown in table 4.3 that is, 5 %, 10 %, 20 % and 30 %, were, 25, 20, 20 and 10 respectively.

<table>
<thead>
<tr>
<th>Feature model</th>
<th>FM</th>
<th>ECR</th>
<th>RCS</th>
<th>$S_0$</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_{[1-2]}$</th>
<th>$C_1$</th>
<th>$C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword in context IS [81]</td>
<td>25</td>
<td>16.0%</td>
<td>5 %</td>
<td>3.31</td>
<td>0.19</td>
<td>12.69</td>
<td>13.01</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 %</td>
<td>5.55</td>
<td>0.3</td>
<td>7.1</td>
<td>7.52</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>4.19</td>
<td>3.18</td>
<td>2.2</td>
<td>5.7</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 %</td>
<td>10.92</td>
<td>0.68</td>
<td>0.64</td>
<td>1.44</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>Digital video system [80]</td>
<td>26</td>
<td>23.1%</td>
<td>5 %</td>
<td>12.54</td>
<td>0.73</td>
<td>6.91</td>
<td>8.52</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 %</td>
<td>5.05</td>
<td>0.91</td>
<td>2.28</td>
<td>4.34</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>8.12</td>
<td>1.12</td>
<td>3.69</td>
<td>5.82</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 %</td>
<td>11.7</td>
<td>1.22</td>
<td>2.91</td>
<td>7.59</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Reference management [60]</td>
<td>31</td>
<td>45.2%</td>
<td>5 %</td>
<td>1.9</td>
<td>0.56</td>
<td>3.63</td>
<td>4.33</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 %</td>
<td>2.02</td>
<td>0.52</td>
<td>4.66</td>
<td>5.33</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>2.28</td>
<td>1.47</td>
<td>4.95</td>
<td>6.73</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 %</td>
<td>2.8</td>
<td>1.81</td>
<td>1.81</td>
<td>3.82</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>Graph editor [57]</td>
<td>32</td>
<td>23.3%</td>
<td>5 %</td>
<td>1.66</td>
<td>0.13</td>
<td>7.83</td>
<td>8.18</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 %</td>
<td>4.99</td>
<td>1.3</td>
<td>11.62</td>
<td>13.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>13.65</td>
<td>1.03</td>
<td>9.32</td>
<td>10.65</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Web portal [61]</td>
<td>43</td>
<td>25.6%</td>
<td>5 %</td>
<td>1.66</td>
<td>0.41</td>
<td>2.84</td>
<td>3.71</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 %</td>
<td>2.32</td>
<td>1.96</td>
<td>2.75</td>
<td>4.94</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>4.34</td>
<td>2.84</td>
<td>4.33</td>
<td>7.47</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 %</td>
<td>4.2</td>
<td>4.0</td>
<td>3.68</td>
<td>7.95</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Document generation [86]</td>
<td>44</td>
<td>29.5%</td>
<td>5 %</td>
<td>5.31</td>
<td>1.07</td>
<td>21.1</td>
<td>22.6</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 %</td>
<td>3.2</td>
<td>1.11</td>
<td>9.51</td>
<td>10.86</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>7.93</td>
<td>2.03</td>
<td>6.06</td>
<td>8.54</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 %</td>
<td>11.72</td>
<td>3.84</td>
<td>7.01</td>
<td>11.15</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Laptop computer [60]</td>
<td>46</td>
<td>80.4%</td>
<td>5 %</td>
<td>10.35</td>
<td>0.32</td>
<td>68.87</td>
<td>69.57</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 %</td>
<td>12.31</td>
<td>0.5</td>
<td>47.7</td>
<td>48.58</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.3: Time measurements for staged configuration on medium-sized test feature models
4.4. Experimental Evaluation

Analysis

Tests on these models are, perhaps, the first depicting some interesting singularities. For instance, the last test on “Graph editor” feature model and the last two tests on “Laptop computer” feature model are not available; this is a consequence of a high number of features for all the random selections calculated by the computer appliances described previously which are verified during stage zero and none of them succeed.

“Graph editor” has many mandatory features and extra constraints are usually applied on optional or grouped features, hence it is reasonable that a high relative configuration size fails. But “Laptop computer” fails because the extra constraints rate is extremely high even though most of the features are non-mandatory but they are part of XOR feature groups, which limits the number of features that can be selected at the same time.

Another singularity, inherited from the experimentations on small-sized feature models, is the relation between the average time for the first and second stage configurations. While the average time of the first stage tends to increase when the size of the configuration increases, the average time of the second stage tends to do the opposite, that is, decrease. But, as previously noted, this is a consequence of how the algorithms carry out their tasks. While finding a B-path from any node to the root node is relatively easy, doing the opposite is more complicated because there are more possibilities, and when there is a failure (for instance, choosing a feature which conflicts with another one already chosen) the second algorithm has to backtrack to find a suitable and valid feature.
Tests on big-sized feature models

Once again, the test feature models were obtained from SPLOT [60]. As previously said, the sizes of feature models for this set of tests had to be greater than 50 features and some models were discarded because of an ECR = 0%.

The number of random generated configurations according to the RCS values shown in table 4.4, that is, 5 %, 15 %, 25 % and 35 %, were, 50, 30, 20 and 10 respectively.

<table>
<thead>
<tr>
<th>Feature model</th>
<th>FM₅</th>
<th>ECR</th>
<th>RCS</th>
<th>S₀</th>
<th>S₁</th>
<th>S₂</th>
<th>S₁₋₂*</th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arcade game [60]</td>
<td>61</td>
<td>55.7%</td>
<td>5 %</td>
<td>11.36</td>
<td>3.17</td>
<td>21.33</td>
<td>25.03</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>11.05</td>
<td>3.75</td>
<td>7.8</td>
<td>12.1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>25.99</td>
<td>8.6</td>
<td>5.1</td>
<td>14.47</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35 %</td>
<td>40.31</td>
<td>14.7</td>
<td>20.34</td>
<td>35.91</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Home Integration System [42]</td>
<td>67</td>
<td>11.9%</td>
<td>5 %</td>
<td>4.36</td>
<td>1.98</td>
<td>12.66</td>
<td>15.03</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>8.72</td>
<td>4.19</td>
<td>7.78</td>
<td>12.54</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>14.59</td>
<td>1.37</td>
<td>3.17</td>
<td>5.29</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ecological car [60]</td>
<td>94</td>
<td>4.3%</td>
<td>5 %</td>
<td>9.04</td>
<td>3.52</td>
<td>25.59</td>
<td>29.65</td>
<td>0.72</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>19.71</td>
<td>14.13</td>
<td>0.0</td>
<td>14.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>28.13</td>
<td>12.41</td>
<td>0.0</td>
<td>12.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Electronic shopping [55]</td>
<td>287</td>
<td>11.8%</td>
<td>5 %</td>
<td>39.79</td>
<td>28.58</td>
<td>287.98</td>
<td>317.93</td>
<td>1.0</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15 %</td>
<td>77.1</td>
<td>55.29</td>
<td>188.74</td>
<td>245.71</td>
<td>1.0</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 %</td>
<td>124.2</td>
<td>79.46</td>
<td>147.8</td>
<td>229.73</td>
<td>1.0</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35 %</td>
<td>199.17</td>
<td>110.97</td>
<td>106.54</td>
<td>219.71</td>
<td>1.0</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 4.4: Time measurements for staged configuration on big-sized test feature models

Analysis

Perhaps these models are the most interesting ones, because of their size being close to real world feature models, specially the “Electronic shopping” feature model. First two models, “Arcade game” and “Home Integration System”, which are of similar size but quite a different ECR rate, have a contrasting time performance when comparing the RCS. While both have a higher time cost for small selections (5 %), the tendency of the former is to increase the overall staged configuration time but the contrary happens for the latter; this is directly related to the number of features involved in extra constraints (more features, more hyper-arcs, which in the end will notably increase the cost of compatibility checks when looking for partial configurations).

Last model, “Electronic shopping”, summarises all tests. It shows how the pre-process and first stage algorithms proportionally increase their times as the selection size increases, and the second stage algorithm is much more efficient when the staged configuration from the first phase has more features selected because most of the minimum cardinalities of grouped features will have already been satisfied. Overall efficiency of a selection such that RCS ≤ 35% seems reasonable given that S₀₋₂* is below 400 milliseconds (on average).
4.4. Experimental evaluation

Tests on benchmark feature models

Table 4.5 illustrates a few results of the set of benchmark tests, obtained from SPLOT [60] for consistent randomised feature models of 500 and 1000 features and an ECR of 10 %.

The number of random generated configurations according to the RCS values shown in the table, that is, 1 %, 2 %, 5 %, were, 200, 150 and 100 respectively.

<table>
<thead>
<tr>
<th>Feature model</th>
<th>FM</th>
<th>ECR</th>
<th>Configuration</th>
<th>RCS</th>
<th>S₀</th>
<th>S₁</th>
<th>S₂</th>
<th>S(r−2)</th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLOT-FM-50-SAT-1</td>
<td>517</td>
<td>9.6 %</td>
<td>69.68</td>
<td>7.72</td>
<td>3544.49</td>
<td>3553.85</td>
<td>0.98</td>
<td>0.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLOT-FM-50-SAT-2</td>
<td>511</td>
<td>9.4 %</td>
<td>71.16</td>
<td>5.71</td>
<td>3115.95</td>
<td>3123.34</td>
<td>0.99</td>
<td>0.34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLOT-FM-50-SAT-3</td>
<td>518</td>
<td>9.8 %</td>
<td>70.56</td>
<td>6.75</td>
<td>2718.52</td>
<td>2726.86</td>
<td>0.95</td>
<td>0.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLOT-FM-100-SAT-1</td>
<td>1034</td>
<td>9.6 %</td>
<td>255.7</td>
<td>21.15</td>
<td>18456.9</td>
<td>18481.16</td>
<td>0.78</td>
<td>0.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLOT-FM-100-SAT-2</td>
<td>1036</td>
<td>9.4 %</td>
<td>217.36</td>
<td>23.83</td>
<td>15786.21</td>
<td>15812.61</td>
<td>0.74</td>
<td>0.38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLOT-FM-100-SAT-3</td>
<td>1037</td>
<td>9.4 %</td>
<td>217.3</td>
<td>25.35</td>
<td>16636.54</td>
<td>16664.67</td>
<td>0.81</td>
<td>0.16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Time measurements on benchmark feature models

Analysis

Results show how the complexity of any verification and configuration is concentrated on the first stage, $S_1$. It can be argued that the number of selected features is low, but experiments (not included here) with higher number of selected features showed no partial configurations (first stage), which is a sign that the feature model is quite complex and the extra constraints makes it even more complex. Proof of this complexity is the average ratio of configurations per stage, $C_1$ and $C_2$.

The high average time cost for the second stage, $S_2$, for instance close to 17 seconds when RCS is 1 % in “SPLOT-FM-100-SAT-3” feature model is an evidence of how selecting feature can trigger the examination of its subtree because it has to be completed as there are mandatory features in between. Note how a low $S_2$ is related to a very low $C_2$, and indication that many configurations were discarded at early iterations of the completion stage.

More feature models were tested, but since the results are similar a representative sample is depicted.
Multiple parent features tests on a customised feature model [55]

Last set of tests show the feasibility of feature diagram modelling as hyper-graphs by randomly adding (structural) hyper-arcs to features so they have more than one parent. This approach will cause multiple variants while carrying out the first-stage configuration (one variant per parent feature if the feature with multiple parents is selected, and possibly combinations of other features with multiple parents if they are selected too), and each of these partial configurations will have to be completed.\(^{15}\)

In order to make the tests more “real”, the “Electronic shopping” [55] feature model was chosen for these tests. Table 4.6 differs from previous ones in the first column which depicts the customisations introduced, where \(F_2\) and \(F_3\) are the number of features with two and three feature parents respectively. The number of random generated configurations per RCS was not changed.

<table>
<thead>
<tr>
<th>CUSTOMISATIONS</th>
<th>FM(_5)</th>
<th>ECR</th>
<th>RCS</th>
<th>(S_0)</th>
<th>(S_1)</th>
<th>(S_2)</th>
<th>(S_{[1\ldots2]})</th>
<th>(C_1)</th>
<th>(C_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F_2 = 2 \land F_3 = 0)</td>
<td>287</td>
<td>11.8%</td>
<td>5</td>
<td>44.33</td>
<td>57.26</td>
<td>346.2</td>
<td>405.2</td>
<td>1.36</td>
<td>1.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>85.6</td>
<td>147.34</td>
<td>358.66</td>
<td>509.93</td>
<td>1.83</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>135.71</td>
<td>189.38</td>
<td>281.41</td>
<td>474.49</td>
<td>1.95</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35</td>
<td>209.7</td>
<td>205.67</td>
<td>216.59</td>
<td>426.25</td>
<td>1.9</td>
<td>0.9</td>
</tr>
<tr>
<td>(F_2 = 2 \land F_3 = 1)</td>
<td>287</td>
<td>11.8%</td>
<td>5</td>
<td>47.45</td>
<td>59.96</td>
<td>384.5</td>
<td>447.08</td>
<td>1.36</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>85.27</td>
<td>266.1</td>
<td>735.23</td>
<td>1006.87</td>
<td>3.57</td>
<td>2.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>140.35</td>
<td>470.29</td>
<td>881.43</td>
<td>1363.0</td>
<td>6.0</td>
<td>3.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35</td>
<td>229.68</td>
<td>559.04</td>
<td>829.98</td>
<td>1404.13</td>
<td>6.6</td>
<td>4.6</td>
</tr>
<tr>
<td>(F_2 = 2 \land F_3 = 3)</td>
<td>287</td>
<td>11.8%</td>
<td>5</td>
<td>47.32</td>
<td>218.84</td>
<td>1654.22</td>
<td>1881.47</td>
<td>4.58</td>
<td>6.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>95.34</td>
<td>1303.77</td>
<td>4469.7</td>
<td>5810.7</td>
<td>20.77</td>
<td>12.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>152.26</td>
<td>3240.44</td>
<td>8502.7</td>
<td>11836.78</td>
<td>45.95</td>
<td>26.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35</td>
<td>254.95</td>
<td>3762.53</td>
<td>10133.82</td>
<td>14053.74</td>
<td>57.6</td>
<td>43.2</td>
</tr>
<tr>
<td>(F_2 = 3 \land F_3 = 3)</td>
<td>287</td>
<td>11.8%</td>
<td>5</td>
<td>42.65</td>
<td>216.88</td>
<td>1680.09</td>
<td>1903.33</td>
<td>4.56</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>89.61</td>
<td>1473.03</td>
<td>5867.12</td>
<td>7380.1</td>
<td>22.73</td>
<td>18.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>151.83</td>
<td>3737.02</td>
<td>10166.15</td>
<td>14012.18</td>
<td>56.65</td>
<td>31.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>35</td>
<td>251.19</td>
<td>8595.86</td>
<td>30783.49</td>
<td>39636.66</td>
<td>115.2</td>
<td>97.2</td>
</tr>
</tbody>
</table>

Table 4.6: Time measurements on customised test feature model [55] for multiple parent features

Analysis

Given the results, it is quite clear that only one feature with more than one parent increases rapidly the average time cost for both stages (note that phase \(S_0\) suffers very little when compared between the customisations and without customisations, although this is expected because during there is not a backward traversal which is the one which takes care of the multiple partial configurations triggered by features with more than one parent feature),\(^{15}\)

\(^{15}\)Reader shall not forget that every time a feature is added via a require constraint, the first-stage configuration algorithm is triggered and if a feature with multiple parents belongs to the hyper-path between the required and the root features because of the constraint or because it is directly required by a constraint, again, multiple partial configurations will be created\(^{16}\).

\(^{16}\)This is not a mistake, the meaning of \(C_2 > C_1\) is that completing a partial configuration included the selection of a feature with multiple parent features, which triggers more partial configurations that must be completed as well.
but it is also interesting to note that the number of valid configurations increases notably, and the overall efficiency of a selection such that $RCS \leq 35\% \land F_2 = 2, F_3 = 3$ is even better than the same test feature model without customisations, because $S_{[0,2]}$ is below 350 milliseconds (on average), which is a slight improvement.

Other combinations of $RCS, F_2$ and $F_3$ show a similar average time, although the average time measurements seem to increase by 50 milliseconds, though this is a reasonable cost given the benefits (more valid configurations).
CHAPTER 4. VERIFICATION OF FEATURE DIAGRAMS

Overall analysis

To conclude this evaluation it is of great importance to analyse all experimentations altogether because there are some conclusions which can be derived from this sort of overall review and they can not be ignored regardless of the consequences.

On average it seems that the algorithms derived from [Gallo et al.] are quite good in terms of performance for feature models with a reasonable number of features and a fair extra constraints rate, as the average configuration times on “Electronic shopping” feature model (both customised and untouched versions) show, which approximately around 400 ms. This feature model is considered by the author of this thesis a representative model because the model has many similarities in terms of complexity of a previous analysis of another domain (associationism phenomenon) which resulted in a feature model with a similar number of features [74].

There can not be a feature model with a high number of features and a high extra constraints rate because not only it would limit the number of possible configurations but it would also ease the traversal of the feature diagrams. An example of this behaviour is the “Laptop computer” feature diagram, which has a substantial number of features and a high ECR, but because of this ECR, a configuration with many features (which presumably will slow the traversal during the first and specially the second staged configurations) is not possible due to incompatibilities between the selected features.

It can be argued that most of the tested models are trees and there are optimised algorithms to perform such tasks; but the algorithms evaluated in this section show how an implementation of hyper-graph traversal algorithms for feature diagrams configurations does not suffer from the underlying data structure, that is, the traversal algorithms are equally good when traversing a tree or a hyper-graph.

Moreover, when analysing the performance it is important to make sure that tests models are suitable for testing, and this is the reason that many feature models available at SPLOT [60] were discarded, as they were considered to be insufficient for an algorithmic evaluation.

In the end, these experimentations show that hyper-graph feature diagram modelling is feasible, and achieves the same results as other approaches. Perhaps these prototype implementations are not as fast as these other approaches, but the results are promising and encourage further development of optimisations on the algorithms as well as the underlying hyper-graph model as the technique is clearly easier and simpler than others, which should be the goal of any procedure or method.

\[17\] Modelled as a hyper-graph
Chapter 5

Conclusions and future work

“The value of a man resides in what he gives and not in what he is capable of receiving”

Albert Einstein

Hyper-graphs are an abstract mathematical representation to depict associations between members of a finite set of elements. This level of abstractness contributes with some capabilities that would not be possible to benefit from another kind of abstraction, and evidence of this affirmation is the fact that computer science has been using a particular case of hyper-graphs, graphs, during many decades.

Formalising a new underlying data structure for feature diagram modelling is not a trivial task and it requires quite a lot of effort to come up with all the details, characteristics and rigorous methodologies so that it is as robust and efficient as the underlying data structures, trees or directed acyclic graphs, considered in most of the scientific literature about feature modelling.

Nonetheless, the semantic definition of feature diagrams modelled as hyper-graphs, proposed by Laguna and Marqués in [54] and analysed and described in this thesis, shows that it is a simpler and effortless simplification of a feature diagram capable of integrating all elements of a feature model instead of using other procedures such as converting the model to algebraical expressions and using (constraint) solvers on these.

The empirical assessment of the two-phases configuration algorithms described in chapter [4] and based on the proposed traversal algorithms by Gallo et al. in [30], show how the hyper-graph approach is feasible in terms of time and performance, as the results seem to be under reasonable limits (an average time of less than 450 milliseconds to find a valid configuration given a feature model of almost 300 features and a selection of nearly 100 features) for an appropriate user experience.

Furthermore, the results of the prototype implementation of these algorithms not only suggest that the approach is promising but also show how the preprocessing of a feature diagram is a major characteristic to be considered in verification of configuration algorithms because it will notably lower the average times when looking for partial configurations and completing them, but it also simplifies the complexity of these algorithms (for instance, given a feature model, paths between each feature and the root feature can be calculated in advance, and reused later).

However, the implementation of these algorithms has shown minor weaknesses regarding the definition of the extra constraints of a feature model, which lead to an increase of the complexity of the implementation. Even so, these defects are to be expected and can be easily corrected, for instance, preprocess is one possibility.

This thesis has not only illustrated another empirical use of hyper-graphs, which graphs are derived from, in the emerging area of software product line development, but it has also introduced a novel approach to represent in a single model as opposed to other approaches, the most popular decision models in software product line engineering, feature diagrams, towards a better implementation in Computer Aided Software Engineering tools which also requires systematic and feasible algorithms aimed at helping with the configuration process rather than merely indicating the validity of a (user) selection or not.
5.1 Future work

The last chapter can not be finished, neither can this work, without without some proposals for future work that could continue with the research carried out in the time that this thesis has been accomplished, because the completion of a thesis shall not ever be completely ceased for the same reason that research never stops, specially in the area that this work is originated.

It follows some future work suggestions which were considered of interest:

- **Hyper-arc costs/weights**

  A key aspect of the second stage configuration, the completion process, is the policy regarding which features are to be chosen in order to meet the minimum cardinality of those grouped features which must be completed. Although there are other approaches such as choosing the maximum feasible number of features. But every time a feature is chosen, that is, a decision is made, there is a consequence. Therefore, it seems reasonable exploring the possibilities that using costs (or weights) for each feature may lead to, because in every feature model it is reasonable thinking that non-mandatory features do not have the same usefulness or value, and this significance might even differ for each stakeholder.

  Currently, there is work in progress advancing on this alternative procedure for feature diagram’s completion.

- **Aiding tools**

  Although the algorithms proposed in this thesis were able to be tested because there was a good repository of test feature models, SPLOT \[60\], it would have been desirable that a generation tool of randomised (and valid) feature models had been available.

  However, it is rather contradictory because achieving a tool with these characteristics would need if not all parts of the algorithms, at least the essential ones.

- **Synergy with other projects**

  Although the aim of this thesis was not the development of any tool, the ultimate goal of the feature diagram modelling approach as a hyper-graph as well as the traversal algorithms is the utilisation in software product lines, specifically in the configuration phase.

  The hyper-graph modelling and its implementation in configuration tools may benefit from storing the pre-processed diagrams instead of doing calculations on the fly or designing and implementing tools for these tasks. An interesting mechanism is a database, and one interesting project is *HyperGraphDB* \[45\], which is a “general purpose, extensible, portable, distributed, embeddable, open-source hyper-graph database”.

---

68


69
http://www.springerlink.com/content/bk2vkhyfrly9a67h/

http://subs.emis.de/LNI/Proceedings/Proceedings143/article2394.html

http://www.springerlink.com/content/d37156v127641552/

http://staff.lero.ie/lchen/publications/splc09.pdf

http://books.google.com/books?id=EUZsqueGeXoC


http://books.google.com/books?id=THGFPqgAACAAJ

http://www.sei.cmu.edu/programs/pls/sw-product-lines_05_03.pdf

http://books.google.com/books?id=WNgyWFL_YC

http://books.google.com/books?id=cCZXYQPau4C


http://swen.uwaterloo.ca/~kczarne/Progress_OOPSLA05_Workshop_on_Software_Factories.pdf
   http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.4436&rep=rep1&type=pdf


   http://www.springerlink.com/content/c5hrdrkpqfm80y9v/

   http://www.springerlink.com/content/h853730912574j58/

   http://www.springerlink.com/content/w284t624310w8753/

   http://shootout.alioth.debian.org/


   http://www.springerlink.com/content/awn535w405321948/

   http://www.springerlink.com/content/v940421372n86418/

   http://books.google.com/books?id=1SIiAQAAIAAJ

   http://wwwinfo.deis.unical.it/~frank/Papers/Altru1/sicomp-revised.pdf

   http://www.dpmms.cam.ac.uk/~wtg10/2cultures.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.522&rep=rep1&type=pdf

[38] Hull, Elizabeth; Jackson, Ken and Dick, Jeremy: Requirements engineering. Springer Verlag, 2005.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.7278&rep=rep1&type=pdf

http://www.sei.cmu.edu/reports/90tr021.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.7568&rep=rep1&type=pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.8583&rep=rep1&type=pdf

http://portal.acm.org/ft_gateway.cfm?id=1183270&type=pdf

http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2673028/

http://code.google.com/p/hypergraphdb/

http://books.google.com/books?id=Up1QAAAAAMAAJ

http://books.google.com/books?id=RYCMx6o47pMC

http://www.methodsandtools.com/archive/archive.php?id=45


http://www.sse.uni-essen.de/wms/pubs/272_tr-01-07.pdf

http://cat.inist.fr/?aModele=afficheN&cpsidt=2388950

http://www.imf.au.dk/publs?id=499


http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.650&rep=rep1&type=pdf

http://www.springerlink.com/content/e249f69wve00lh65

http://www.omg.org/spec/UML/2.2/Infrastructure

[69] Pohl, Klaus; Böckle, Günter and Van Der Linden, Frank: Software product line engineering: foundations, principles, and techniques. Springer-Verlag, 2005.
http://books.google.com/books?id=jiTvwsMEgL8C


http://www.oryx-project.org/

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.1653&rep=rep1&type=pdf

http://arxiv.org/abs/1004.1001v1

http://almena.uva.es/record=b1568743-S1*spi


http://www.springerlink.com/content/m59g1kkrgpjrnydu/

http://www.fiit.stuba.sk/~vranic/pub/FMM.pdf

http://books.google.com/books?id=pfjFQgAACAAJ

http://mathworld.wolfram.com/Combinatorics.html

http://books.google.com/books?id=61gtAAAACAAJ


Appendix A

Variability management approaches

“Variability is the law of life, and as no two faces are the same, so no two bodies are alike, and no two individuals react alike and behave alike under the abnormal conditions which we know as disease”

William Osler

Table A.1 illustrates, in chronological order, some of the variability management approaches derived from the first proposal, Feature Oriented Domain Analysis (FODA) [40] (see chapter 1 for an illustration in figure 1.1 depicting these approaches as a map.

However, a much more extended study and systematic review of these proposals among others (not necessarily derived from FODA) was carried out by Chen et al. in [14], for which the reader is encouraged to look through it.

<table>
<thead>
<tr>
<th>YEAR</th>
<th>PROPOSAL</th>
<th>AUTHORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>FODA</td>
<td>Kang et al.</td>
</tr>
<tr>
<td>1998</td>
<td>FORM</td>
<td>Kang et al.</td>
</tr>
<tr>
<td></td>
<td>FRSEB (Feature Reuse-Driven Software Engineering)</td>
<td>Griss et al.</td>
</tr>
<tr>
<td>2001</td>
<td>CONSUL (CONfiguration SUpport Library)</td>
<td>Beuche et al.</td>
</tr>
<tr>
<td>2002</td>
<td>FDL (Feature Description Language)</td>
<td>van Deursen et al.</td>
</tr>
<tr>
<td>2004</td>
<td>CBFM (Cardinality-Based Feature Model)</td>
<td>Czarnecki et al.</td>
</tr>
<tr>
<td></td>
<td>RequiLine (Requirements Engineering Tool for Software Product Lines)</td>
<td>von der Maßen and Lichter</td>
</tr>
<tr>
<td>2007</td>
<td>VFD (Varied Feature Diagram)</td>
<td>Schobbens et al.</td>
</tr>
</tbody>
</table>

Table A.1: FODA derived variability management approaches
Software product line development processes

“An abstraction is one thing that represents several real things equally well”

Edsger W. Dijkstra

The following diagrams illustrate the categorisation established by [Krueger in [49] for software product line development according to the procedure followed by the development team when analysing, designing and implementing the platform of the product line.

A brief description of each process is detailed in section 2.1.4

Figure B.1: Software Product Line’s proactive approach

Figure B.2: Software Product Line’s reactive approach
**APPENDIX B. SOFTWARE PRODUCT LINE DEVELOPMENT PROCESSES**

**Figure B.3:** Software Product Line’s extractive approach

**Figure B.4:** Software Product Line’s refactorive approach
“The aim of science is to seek the simplest explanations of complex facts. We are apt to fall into the error of thinking that the facts are simple because simplicity is the goal of our quest.”

Alfred North Whitehead

Part of the constraint notation used in the sample feature models available at [60] was not compatible with the modeling approach considered for this thesis; therefore, it follows two tables describing those conversions for greater clarity.

Table C.1 illustrates the conversion of conjunctive normal forms of two clauses to equivalent feature diagrams (Czarnecki-Eisenecker notation), graphical and textual constraints. While table C.2 depicts the conversion of conjunctive normal forms of three clauses to equivalent feature diagrams (Czarnecki-Eisenecker notation) and textual constraints.

Given a feature diagram, \( \mathcal{F} = (\mathcal{N}, \mathcal{E}, \mathcal{V}, \delta) \), as defined in 3.2, a textual constraint in tables C.1 and C.2 is formulated as follows:

- **require** \((\mathcal{S}, \mathcal{D}, [\text{min} \ldots \text{max}])\) such that
  - \( S, D \subseteq V' \land S \cap D = \emptyset \land |S| = 1 \land |D| \geq 1 \)
  - \( \text{min}, \text{max} \in \mathbb{Z} \land \text{min} \geq 0, 0 < \text{max} \leq |D|, \text{min} \leq \text{max} \)

- **mutex** \((\mathcal{S}, [\text{min} \ldots \text{max}])\) such that
  - \( S \subseteq V' \land |S| > 1 \)
  - \( \text{min}, \text{max} \in \mathbb{Z} \land \text{min} = 0, \text{max} = 1 \)
### Table C.1: Conversion of two clauses CNF to feature diagram, graphical and textual constraint

<table>
<thead>
<tr>
<th>CNF</th>
<th>Feature diagram</th>
<th>Graphical const.</th>
<th>Textual const.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \lor B$</td>
<td><img src="" alt="Feature Diagram" /></td>
<td>$\text{Root}$ $\langle 1-2 \rangle$</td>
<td>$\text{require}(\lor, [A,B], [1\ldots2])$</td>
</tr>
<tr>
<td>$\overline{A} \lor B$</td>
<td><img src="" alt="Feature Diagram" /></td>
<td>$\text{Root}$ $\langle 0-1 \rangle$</td>
<td>$\text{mutex}(A,B, [0\ldots1])$</td>
</tr>
<tr>
<td>$\overline{A} \lor B$</td>
<td><img src="" alt="Feature Diagram" /></td>
<td></td>
<td>$\text{require}(A,B, [1\ldots1])$</td>
</tr>
<tr>
<td>CNF</td>
<td>FEATURE DIAGRAM</td>
<td>TEXTUAL CONST.</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>$A \lor B \lor C$</td>
<td><img src="" alt="Feature Diagram" /></td>
<td>$require(\lor, [A, B, C], [1...3])$</td>
<td></td>
</tr>
<tr>
<td>$\bar{A} \lor \bar{B} \lor \bar{C}$</td>
<td><img src="" alt="Feature Diagram" /></td>
<td>$require(\neg, [A, B, C], [0...2])$</td>
<td></td>
</tr>
<tr>
<td>$\bar{A} \lor B \lor C$</td>
<td><img src="" alt="Feature Diagram" /></td>
<td>$require(A, [B, C], [1...2])$</td>
<td></td>
</tr>
<tr>
<td>$A \lor \overline{B} \lor \overline{C}$</td>
<td><img src="" alt="Feature Diagram" /></td>
<td>$mutex([A, require(\overline{BC}, [B, C], [0...1]), [0...1]])$</td>
<td></td>
</tr>
</tbody>
</table>

Table C.2: Conversion of three clauses CNF to feature diagram and textual constraint
Classical graph traversal algorithms

“Mathematics is the language of size, shape and order and that it is an essential part of the equipment of an intelligent citizen to understand this language”

Lancelot Hogben

Some classical traversal algorithms for graphs are listed below (pseudocode). Most of them are designed for undirected graphs; however, they can be easily customised for directed graphs.

- **Breadth-first search (BFS)**

- **Depth-first search (DFS)**
  - Depth-first search for directed acyclic graphs (DAG-DFS)

- **Single-source shortest path (SSSP)**
  - Dijkstra (Dijkstra)
  - Single-source shortest path for directed acyclic graphs (DAG-SSSP)

```plaintext
procedure bfs(G = (V, E), r)
for each v ∈ V do
    visited[v] := false;
    d[v] := ∞;
    p[v] := nil;
end for
visited[r] := true; Q := 0; d[r] := 0;
enqueue(r, Q);
while not empty(Q) do
    v := dequeue(Q);
    // Each vertex stores an adjacency list of its outgoing edges
    for each w ∈ adj(v) do
        if not visited[w] then
            p[w] := v;
            d[w] := d[v] + 1;
enqueue(w, Q);
        end if
    end for
end while
end procedure
```

Listing D.1: Breadth-first search procedure [1]
1 procedure dfs(G = (V,E))

3 visited[v] := true; // Not visited vertex v has been discovered
4 time := time + 1;
5 dis[v] := time;

7 // Each vertex stores an adjacency list of its outgoing edges
8 for each w ∈ adj(j) do
9     if not visited[w] then
10         p[w] := v;
11         dfs(w); // Explore edge (v,w)
12     end if
13 end for
15 finished[v] := true;
16 time := time + 1;
17 fin[v] := time;
19 end procedure

Listing D.2: Depth-first search procedure [1]

1 procedure dag-dfs(G = (V,E))

3 for each v ∈ V do
4     visited[v] := false;
5     finished[v] := false;
6     P[v] := nil;
7 end for

9 // Global variable used for timestamping
10 time := 0;

12 for each v ∈ V do
13     if not visited[v] then
14         dfs(v);
15     end if
16 end for
18 end procedure

Listing D.3: Directed depth-first search procedure for directed acyclic graphs [1]
\begin{verbatim}
1 procedure dijkstra(G = (V, E), w, s)
3 // Initialize single source s of G
4 for each v ∈ G do
5     d[v] := ∞;
6     pi[v] := nil;
7 end for
8 d[s] := 0;
10 S := ∅;
11 Q := V;
13 while Q ̸= ∅ do
14     u := select(Q);
15     S := S ∪ {u};
16     for each v ∈ adj(u) do
17         // relax(u, v, w)
18         if d[v] > d[u] + w(u, v) then
19             pi[v] := u;
20         end if
21     end for
22 end while
24 end procedure

Listing D.4: Dijkstra’s procedure \[19\]
\end{verbatim}

\begin{verbatim}
1 procedure dag-sssp(G = (V, E), w, s)
3 // Topological sort of G, topological_sort(G)
4 dfs(G); // Compute finishing time, fin[v], for each vertex v
6 // Initialize single source s of G
7 for each v ∈ G do
8     d[v] := ∞;
9     pi[v] := nil;
10 end for
11 d[s] := 0;
13 // For each vertex u, taken in topologically sorted order
14 for each u ∈ topological_sort(G) do
15     for each v ∈ adj[u] do
16         // relax(u, v, w)
17         if d[v] > d[u] + w(u, v) then
18             pi[v] := u;
19         end if
20     end for
21 end for
23 end procedure

Listing D.5: Single-source shortest path procedure for directed acyclic graphs \[19\]
\end{verbatim}