THESIS / THÈSE

DOCTOR OF SCIENCES

Quality of feature diagram languages: formal evaluation and comparison

Trigaux, Jean-Christophe

Award date:
2008

Awarding institution:
University of Namur

Link to publication

FUNDP - PRECISE

University of Namur

Faculté d'Informatique

Rue Grandgagnage, 21

B - 5000 NAMUR (Belgique)

Quality of Feature Diagram Languages:

Formal Evaluation and Comparison

Jean-Christophe Trigaux

Thèse présentée en vue de l'obtention du titre de
Docteur en Sciences

September 2008

Jury: Prof. Dr. Jean-Marie Jacquet (Président), University of Namur,
Prof. Dr. Patrick Heymans (Promoteur), University of Namur,
Prof. Dr. Pierre-Yves Schobbens (co-Promoteur), University of Namur,
Prof. Dr. Thierry Massart, Université Libre de Bruxelles,
Prof. Dr. David Benavides, University of Sevilla,
Prof. Dr. Wim Vanhoof, University of Namur.

Abstract

In software engineering, software reuse has been a popular topic since 1968. Nowadays, Software Product Line (SPL) engineering promotes systematic reuse throughout the whole software development process. Within SPL, reusability strongly depends on variability. In this context, variability modelling and management are crucial activities that crosscuts all development stages. Different techniques are used to model variability and one of them is Feature Diagrams (FDs). FD languages are a family of popular modelling languages used to model, and reason on, variability. Since the seminal proposal of a FD language, namely FODA, many extensions have been proposed to improve it. However, the pros and cons of these languages are difficult to evaluate for two main reasons: (1) most of them are informally defined and (2) no well defined criteria were used to justify the extensions made to FODA. As a consequence, variability modelling and management techniques proposed in the literature or used by practitioners may be suboptimal.

Globally, this work underlines that the current research on FDs is fragmented and provides principles to remedy this situation. A formal approach is proposed to introduce more rigour in the motivation, definition and comparison of FD languages. Thereby, examining their qualities should be more focused and productive. A formal approach guarantees unambiguity and is a prerequisite to define formal quality criteria and to produce efficient and safe tool automation. A quality analysis is necessary to avoid the proliferation of languages and constructs that are an additional source of misinterpretations and interoperability problems. The creation or selection of a FD language should be driven and motivated by rigourous criteria. Translations from one FD language to another should be defined and carefully studied to avoid interoperability problems.

The main contributions of this work are: (1) to use a quality framework to serve as a roadmap to improve the quality of FD languages, (2) to formally evaluate and compare FD language qualities according to well-defined criteria and following a clear method, (3) to formally define and motivate a new FD language that obtains the best scoring according to the quality criteria and (4) to develop tool support for this language.

Keywords

Software Product Line, Requirements Engineering, Formal Methods, Formal Semantics, Variability Management, Feature Diagrams, Feature Modelling, Automated Reasoning, Language Evaluation, Language Quality.

II

Acknowledgement

First of all, I express my gratitude to all members of the Jury for their helpful and constructive comments.

I would like to thank, Prof. Klaus Pohl and all the SSE research lab for their hospitality during my stay in Essen.

Special thanks go to Prof. Patrick Heymans and Prof. Pierre-Yves Schobbens for their invaluable supervision, their time, their communicative enthusiasm for research and their personal involvement in both my PhD thesis and research project. Furthermore, I would like to thank the members of the PReCISE research centre and especially, Michael, Yves, Raimundas, Germain, Andreas and Arnaud. It was a pleasure to share the same office and to work in a stimulating and friendly atmosphere. I have enjoyed our fruitful discussions on various topics going from research to leisure. You are a continuous source of motivation, inspiration and amusement. Thank you also to all other past and present colleagues.

I would also like to thank my family and friends for their generosity and support.

Finally, my sincere thank goes to Laurence for supporting me all the way through, for her continuous love, encouragement and comprehension.

Jean-Christophe Trigaux

IV

Contents

List of Tables

List of Figures

Abbreviations	Definitions
ASF	Algebraic Specification Formalism (van Deursen and Klint, 2002)
BC	Boolean Circuits
BDD	Binary Decision Diagrams
BFT	Batory Feature Tree language
CASE	Computer-Assisted Software Engineering
CD	Class Diagrams
CFD(OP)	Constraintless Feature Diagram language
CFT(OP)	Constraintless Feature Tree language
CR	Composition Rules
COFD	Constraintless Original FORM Feature Diagram language
COFT	Constraintless Original FODA Feature Tree language
CRFD	Constraintless RSEB (with or-nodes) Feature Diagram language
CRFT	Constraintless Feature RSEB Tree language
DAG	single-rooted Directed Acyclic Graph
EFD	Extended Feature Diagram language
EQBF	Existentially Quantified Boolean Formulae
FD	Feature Diagram
FFD	Free Feature Diagram language
FODA	Feature Oriented Domain Analysis (Kang et al., 1990)
FoFD	Forfamel Feature Diagram language
FOPLE	Feature-Oriented Product Line Engineering (Kang et al., 2002)
FORM	Feature-Oriented Reuse Method (Kang et al., 1998)
GCT	Graphical Constraint Type
GMF	Graphical Modelling Framework
GP	Generative Programming (Eisenecker and Czarnecki, 2000)
GPFT	Generative Programming Feature Tree language
GT	Graph Type
ITG	Iterative Tree Grammar

Abbreviations	Definitions
$\mathcal{L}_X$	Syntactic domain of a language X
MDE	Model-Driven Engineering
mutex	MUTually EXclusive with
$\mathcal{M}_X$	Semantic function of a language X
NF	Normal Form
NLC	Node-labelled Controlled (Janssens and Rozenberg, 1980)
NLDAG	Node-Labelled Directed Acyclic Graph
NT	Node Type
NP	Nondeterministically Polynomial (Papadimitriou, 1994)
OCL	Object Constraint language
OFD	FORM Feature Diagram language
OFT	FODA Feature Tree language
OP	Node type OPerator
PLUSS	Product Line Use case modelling for System and Software engineering (Eriksson et al., 2005)
PFT	PLUSS Feature Tree language
PL	Product Line
Pr	Propositional Logic
QBF	Quantified Boolean Formulae
RFD	RSEB Feature Diagram language
RSEB	Reuse-Driven Software Engineering Business (Griss et al., 1998)
$\mathcal{S}_X$	Semantic domain of a language X
SAT	Boolean SATisfiability (Papadimitriou, 1994)
SDF	Syntax Definition Formalism (van Deursen and Klint, 2002)
SPL	Software Product Line
SPLE	Software Product Line Engineering
TCL	Textual Constraint Language
UML	Unified Modelling Language (OMG, 2008)
VBFD	van Gurp and Bosch Feature Diagram language (van Gurp et al., 2001)
vDFD	van Deursen and Klint Feature Diagram language (van Deursen and Klint, 2002)
VFD	Varied Feature Diagram language
XFD	XML-based Feature Diagram language
XOR	Exclusive OR

Introduction

In software engineering, software reuse (Jacobson et al., 1997) has received a substantial attention by the research community since the publication of the seminal papers: *Mass Produced Software Components* (McIlroy, 1968) and *On the Design and Development of Program Families* (Parnas, 1976).

Nowadays, many researchers and organisations still believe in software reuse as a powerful means to improve productivity and quality in software development. Software reuse is defined by (Krueger, 1992) as *"using existing artifacts during the construction of a new software system"*. Consequently, it helps avoiding redevelopment and capitalising on previous work. Hence, well-established solutions can be deployed in new contexts with minimal efforts, leading to better products delivered in shorter times with reduced development and maintenance costs.

Software reuse's advantages have been widely acknowledged (Biggerstaff and Perlis, 1989a,b), however, at that time, significant improvements were difficult to deliver (Krueger, 1992; IEEE Software Staff, 1994). In the past decades, several development paradigms have proposed approaches to improve software reuse. Many concepts have been introduced to deal with reuse (see Table 1) at various software development phases.

Software reuse has often been limited to principles and mechanisms that facilitate reuse within the classical software development process. Nevertheless restricting reuse to code and managing it application by application does not sufficiently pay off. Lately, planned and systematic software reuse has been considered as the focus of a particular and novel software development process with the following qualities:

- it enables and advocates the development of sets of applications that share common functionalities (commonalities),

- its efficiency relies on the number of applications developed and on the degree of commonality they share,

- it manages reuse through all development phases including requirements, design, code and tests,

- it industrialises software development.

This new software development process is known as Software Product Line Engineering (SPLE) where a *Software Product Line* (SPL) is defined as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or*

1

Decade	Development Paradigm	References	Reusable artifacts	Phases
1970s	Procedural	(Sebesta, 1993)	Modules	Coding
1980s	Object-Oriented	(Dahl and Nygaard, 1966; Henderson-Sellers, 1992; Armstrong, 2006)	Classes	Coding Design
1990s	Component-Oriented	(D'Souza and Wills, 1999)	Components	Coding Design
1990s	Agent-Oriented	(Shoham, 1990; Wooldridge and Jennings, 1995)	Agents	Coding Design
1990s	Aspect-Oriented	(Kiczales, 1996)	Aspects	Coding Design
1990s	Software Product Lines	(Foreman, 1996; Clements and Northrop, 2001; Pohl et al., 2005)	Core Assets	All
2000s	Model-Driven	(Kleppe et al., 2003; Thomas and Barry, 2003; Thomas, 2004)	Models	Coding Design

Table 1: Software Reuse over the years

mission and that are developed from a common set of core assets in a prescribed way" (Clements and Northrop, 2001).

SPLE introduces the concept of *core assets* that generalises all reusable artifacts proposed in the previous approaches (see Table 1). *Core assets* refer to pieces of code (modules, classes, components, aspects, models) but are not limited to them. In addition, they include requirement documents, architecture descriptions, test cases and artifacts from all phases of the single software development life-cycle.

As illustrated in Figure 1, the classical (i.e. waterfall) single software development process encompasses knowledge, tools and methods to define application requirements and then to perform application design, coding and testing. Many extensions of this process have been proposed. SPLE suggests to adapt it in order to leverage reuse in a systematic manner.
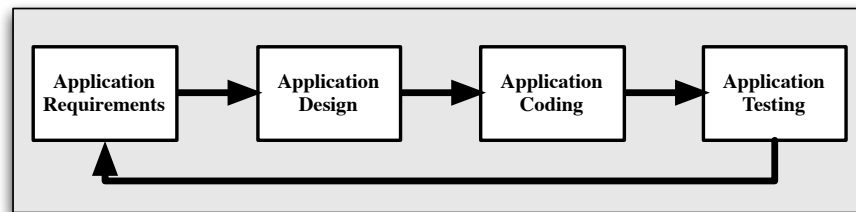
Figure 1: Software Development Life-Cycle

SPLE promotes *systematic reuse* and intends to maximise it. In current practice however, reuse is frequently *opportunistic* and essentially lacks systematisation and standardisation. *Opportunistic reuse* is a non-repeatable process where various realisation mechanisms are used in an ad-hoc manner. Examples of realisation mechanisms are: cut and paste, inheritance, frameworks, design-patterns, software libraries, components, agents, aspects, web services. While it may suffice within limited developer teams working on small projects involving few stakeholders, when projects grow in size with more products and higher complexity, systematic reuse is strongly recommended. Planning, systematising and institutionalising reuse throughout the whole software development life-cycle is crucial. Ultimately, the SPL would benefit from large scale economies thereby reducing its costs and improving its productivity, time to market and software quality.

One of the most challenging aspects of systematic reuse is **variability management**, i.e., how to describe, manage and implement the commonalities and variabilities existing between SPL members. Indeed, each member shares commonalities with all SPL members and differs from them through variabilities. Furthermore, variability impacts the whole development life-cycle. Variability analysis is therefore essential before constructing any reusable artefact. This global analysis describes what varies in the SPL, how it varies and why it varies. Variability management is a complex endeavour. Therefore, models abstracting variability from other considerations are highly profitable.

Different techniques are used to model variability. One of them is called Feature Modelling (Eisenecker and Czarnecki, 2000) and is dedicated to model variability using **Feature Diagram (FD) languages** (Kang et al., 1990). The main purpose of FD languages is to model variability in terms of "features" at a relatively high level of granularity. FD languages enable (1) to capture common and variable features, (2) to represent dependencies between features, and (3) to determine combinations of features that are allowed or disallowed in a SPL. The notion of *feature* can be understood as *"a distinguishable characteristic of a concept (e.g., system, component and so on) that is relevant to some stakeholder of the concept"* (Eisenecker and Czarnecki, 2000).

The seminal FD language was first introduced by Kang *et al.* back in 1990 as part of the FODA (Feature Oriented Domain Analysis) method (Kang et al., 1990). A simple FD modelling variability in a SPL of car systems (see Figure 2) has been proposed by Kang *et al.* to illustrate the main FD language concepts. This model represents a SPL of car systems. Each car system is a combination of features. Each feature may be decomposed into sub-features. Each combination of features is determined according to the constraints imposed by the FD. For instance, the arc between the features "Manual" and "Automatic" indicates that they are alternative features. It means that both features may not appear at the same time in a combination. Therefore no car may have a manual and an automatic transmission at the same time. Once all the constraints are resolved, four different car systems (combination of features) are allowed. The four possible car systems are:

- a car system with manual transmission and without air conditioning,

- a car system with manual transmission and air conditioning,

- a car system with automatic transmission and without air conditioning,

- a car system with automatic transmission and air conditioning.
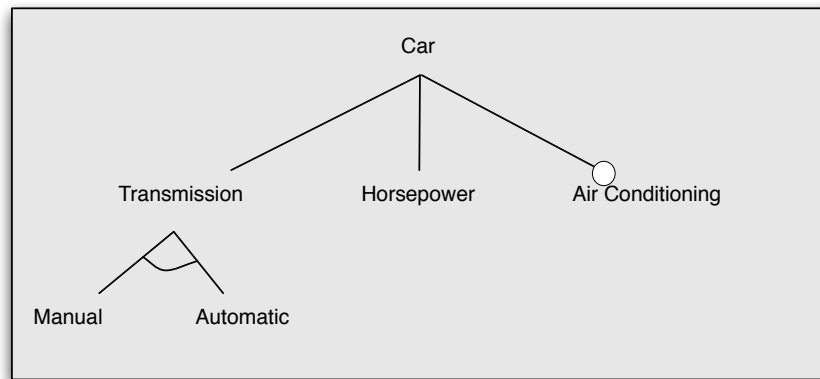
Figure 2: FODA FD: Car Systems (Kang et al., 1990)

Motivation

Feature Modelling is a popular technique that has the potential to deliver significant improvements in terms of variability management, development time, quality and costs. The benefits associated to Feature Modelling are however still restrained. Many reasons may explain this situation. Throughout the thesis we focus our efforts on issues essentially concerning FD languages:

- There is a profusion of FD language proposals: (Kang et al., 1990, 1998; Griss et al., 1998; Eisenecker and Czarnecki, 2000; van Gurp et al., 2001; van Deursen and Klint, 2002; Mannion, 2002; Riebisch et al., 2002; Riebisch, 2003; Cechticky et al., 2004; Eriksson et al., 2005; Czarnecki et al., 2005c,b; Batory, 2005; Benavides et al., 2005a; Wang et al., 2005a,b; Sun et al., 2005; Asikainen et al., 2006; Janota and Kiniry, 2007),

- FD languages are not always properly defined,

- FD languages have not been compared and evaluated according to a clear and systematic method,

- FD languages are not sufficiently supported by tools and, hence, lack automatisation.

Clearly, many different proposals exist for FD languages, what is the most suitable is unclear and no standard exists. Accordingly, our first goal is to analyse and categorise existing FD languages in a systematic way. This categorisation should be elaborated according to a well-defined method and to specific criteria in order to evaluate and compare these languages.

In addition, FD reasoning and automation are frequently underexploited and suboptimal. Accordingly, our second goal is to elaborate safe and efficient tool support based on formal approaches. FD reasoning addresses typical questions that arise in every SPL containing various possible combinations of features (a.k.a products):

- Is a combination of features part of the SPL or not? If not, how should we extend the SPL to include this new combination of features?

- Which and how many combinations of features are valid within the SPL?

- Which combinations of features are invalid and which constraint(s) do they break?

- How many products include a given feature?

- Can we, and if so, how do we translate a FD written in one FD language into another one, preserving the same allowed combinations?

- Are two SPLs described in two different FDs equivalent? If not, can we integrate them, and what SPL will result from this integration?

This thesis aims to improve FD languages' definition, automation and comparison from research and practical perspectives. In terms of research, our purpose is to collect objective knowledge on FD languages and to improve them usefully. In terms of practice, our purpose is to deliver powerful tool support to handle FD languages and to decrease the complexity associated to variability modelling and reasoning. Essentially, we claim that formal approaches and formal quality criteria help to achieve these goals.

Thesis

Currently, formalisation of, and tool support for, FD languages are still immature (Fey et al., 2002a; Batory et al., 2006). However, proper formalisation is a prerequisite for safe and efficient tool support. **Formal approaches** propose various mechanisms to (1) specify the system, (2) formalise the desired properties of it and (3) to prove that the specification satisfies the properties (Bowen and Hinchey, 2006). Used in a proper way, formal approaches enable to provide formal specifications. These specifications are independent of any implementation and assure that the *"what"* should be developed is described rather than the *"how"* it should be developed. In addition, these specifications are computer-understandable, concise, unambiguous and complete. Formal approaches allow identifying ambiguities or errors and eliminating them in the early phases of the development process. Therefore, adopting them has a great potential to improve the quality of FD languages and their tool support.

Similarly, evaluation and comparison of FD languages are also immature and mainly based on informal quality criteria (Djebbi and Salinesi, 2006). **Formal quality criteria** should be based on well established theories such as language theory (Hopcroft et al., 2000) or computational complexity theory (van Leeuwen, 1990; Papadimitriou, 1994) and helps reducing subjectivity when comparing languages. We claim that both formal and informal criteria are complementary and should be combined to depict a better quality panorama.

Hence, our thesis mainly concerns language engineering and the thesis statement can be summarised as follows: Adopting formal approaches and formal quality criteria contributes to better FD languages with more rigourous methods to define and compare them.

Research Problem and Questions

In general, this thesis investigates the problem of quality evaluation and comparison of specific modelling languages called "Feature Diagram (FD) languages". Several research questions are inherent to this problem and provide insights as to why and how we proceed to tackle it. As a starting point for the upcoming reflection and in order to clearly state our research questions, we first describe the main concepts appearing in the title of the thesis: "Quality of Feature Diagram Languages: Formal Evaluation and Comparison". Four concepts must be underlined: (1) Languages, (2) Quality (of Languages), (3) FD Languages and (4) Formal Evaluation and Comparison (of Languages).

- **Languages** refer to systems of symbols used to represent and communicate information (Krogstie, 2001a).

- **Quality (of Languages)** refers to the perception of the degree to which the language meets the users' expectations (Krogstie, 2001a).

- **FD Languages** are a family of popular languages, mostly used to model variability during Requirements Engineering (RE) of Software Product Lines (SPL) (Kang et al., 1990).

- **Formal Evaluation and Comparison (of Languages)** refers to the use of mathematically based techniques for the evaluation and comparison of languages.

Firstly, our investigation is targeted at language quality rather than diagram quality. **Language and diagram qualities** are complementary views. However, in model quality, the study is mainly empirical and the objects of study (the models) are not always easily available. Indeed, real examples of FDs are not accessible mainly due to two reasons: (1) Feature Modelling is an emerging modelling paradigm and (2) FDs represent highly strategic information that companies want to preserve. Therefore, representative (as opposed to research or illustrative) diagrams are almost nowhere to be found. Hence, we focus our efforts on the quality improvement of FD languages for which many references and documentation exist in literature.

Secondly, **quality of languages** itself is a multidimensional and complex endeavour. The notion of user's expectations must be clarified. Indeed, different users may have different expectations for the same language. One could favour the match between the statements of the language and his knowledge whereas another could address the degree to which the language lends itself to automatic reasoning or executability, and so on.

Thirdly, **FD languages** are *visual languages* dedicated to represent variability. Visual means that the symbols manipulated by the languages are mainly graphical ones. Variability models should be easily understandable by non-technical users and be effective to communicate with. They should be supported by tools to minimise tractability and complexity issues. As already mentioned, FD languages are characterised by a profusion of different proposals. Many authors have proposed new FD languages or extended previous ones for their own context with different syntaxes and semantics.

Fourthly, we address the evaluation and comparison of FD Languages from a **formal perspective**. Obviously, this perspective is not sufficient in itself and should be complemented by an "informal" one. For instance, the guarantee that a language by itself helps capturing the right information about the domain can not be formally provided. Furthermore, the notion of "adequate" modelling

language is also relative to the context of the use of the language. Priorities over the quality may differ from one company, or project, or individual to another.

Finally, we believe that one critical issue is to formally compare and evaluate FD languages according to their semantics and to well-defined criteria. The definition of such criteria should be driven by a systematic approach where the studied languages and the criteria are formally defined to avoid any ambiguities or misinterpretations. Therefore, our research problem is stated as follows:

Research Problem: How can the quality of FD languages be formally evaluated and compared? From this research problem, we investigate three main **research questions**:

- *RQ.1: Which qualities could be evaluated for a FD language?*
  This question addresses the concept of quality of FD languages. The various quality dimensions should be identified and related to quality of languages in general. The purpose is to identify and understand the main quality dimensions of FD languages.

- *RQ.2: Which formal approaches facilitate evaluation and comparison of FD languages' quality and how to use them?*
  This question involves the analysis of FD language quality according to formal approaches. Which quality dimensions are involved? Which formal evaluation criteria can help us to compare FD languages? How should such an analysis proceed? What are the main limitations to this analysis?

- *RQ.3: How are existing FD languages evaluated according to this analysis?*
  This question studies existing FD languages and compare them according to formal evaluation criteria.

Throughout this thesis, previous work concerning formal languages and quality of languages is recalled and answers to these research questions constitute the core of our contributions.

Claimed Contributions

In general, this thesis underlines that the current research on FDs is fragmented and therefore provides principles to remedy this situation. A formal approach is presented and designed to introduce more rigour in the motivation, definition and comparison of FD languages. Hence, examining their qualities will be more focused and productive. This quality analysis is necessary to avoid the proliferation of languages and constructs that are an additional source of misinterpretations and interoperability problems. In addition, the creation or selection of a FD language should be driven by rigourous criteria.

The main contributions of this thesis are:

- A semantics is defined and discussed for FDs (Bontemps et al., 2004, 2005). This definition is original in the sense that it is formally defined and it is generic to allow language engineers to provide formal definitions for a family of FD languages. This definition of a family of FD languages is named FFD, standing for Free Feature Diagram.

- A method based on formal criteria is described to evaluate FD language quality from a formal perspective (Schobbens et al., 2006; Heymans et al., 2008).

- This method is applied to both informal and formal FD languages (Trigaux et al., 2006; Schobbens et al., 2006, 2007; Heymans et al., 2008). The practical contributions when applying this method are to (1) provide a formal semantics to informal FD languages, (2) compare and discuss FD languages and their constructs according to formal criteria and (3) compare, relate and discuss different proposals of semantics for FD languages including our.

- A new FD language, called Varied Feature Diagram language (VFD), is formally defined using FFD. VFD is proposed as a minimal language with the best rankings according to the defined criteria.

- A Reasoning Tool is proposed to support VFD and to resolve associated decision problems.

Structure

The thesis is organised in five parts and eleven chapters (Figure 3).

Part I, The Research Domain intends to provide enough background knowledge to familiarise readers with the domain to which this thesis contributes. This part contains two chapters.

- **Chapter 1:** *Software Product Lines* introduces the SPL paradigm. Variability is presented as a key issue that should be addressed during the whole SPL process, and in particular during Requirements Engineering.

- **Chapter 2:** *Feature Modelling: The Basic Concepts* introduces Feature Modelling as a way to model variability in SPL, and more specifically the variability in requirements.

Part II, Quality of Models and Languages presents the notion of quality in the context of Models and Languages. This part contains three chapters.

- **Chapter 3:** *A Quality Framework* gives an overview of a global semiotic quality framework assessing the quality of models and languages named SEQUAL and developed in (Krogstie, 2001a). In addition, SEQUAL is refined to evaluate the quality of FD languages according to the formal criteria defined in Chapter 5.

- **Chapter 4:** *Languages: Formal Definition* recalls the basic principles developed by (Harel and Rumpe, 2004) to formally define the abstract syntax and semantics of languages.

- **Chapter 5:** *Languages: Formal Criteria and Quality* presents the set of formally defined criteria proposed to assess the semantically related qualities of languages. These criteria are situated according to the quality framework presented in Chapter 3 and their definition is based on the principles presented in Chapter 4.

Part III, Quality of Feature Diagram Languages presents how the principles associated to language quality could be transposed to FD language quality. This part contains four chapters.
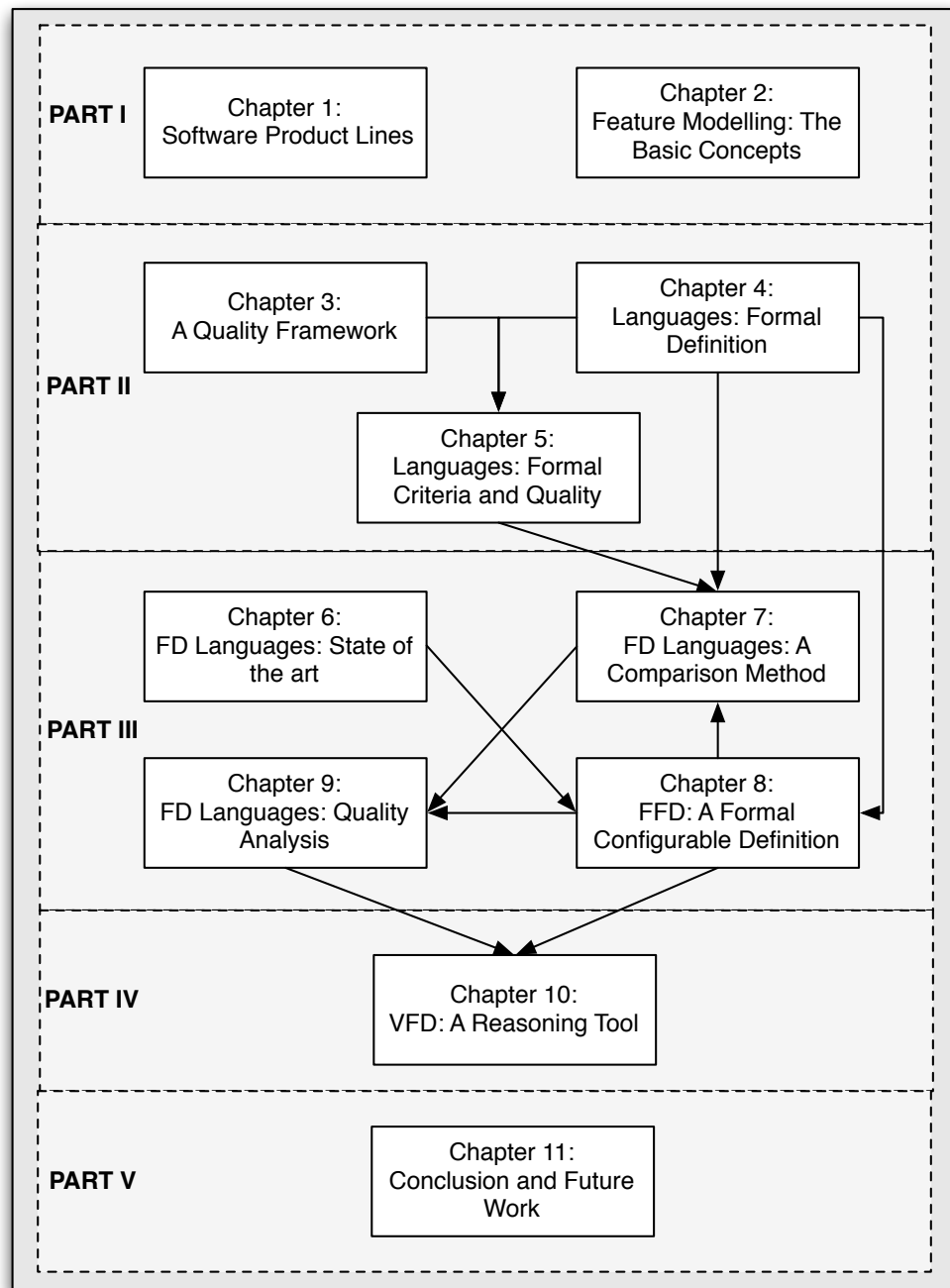
Figure 3: Structure of the Thesis

- **Chapter 6:** *Feature Diagram Languages: State of the Art* surveys the existing FD languages, whether informally or formally defined.

- **Chapter 7:** *Feature Diagram Languages: A Comparison Method* describes and illustrates methods to evaluate and semantically compare FD languages. The principles presented in Chapter 4 are adapted in order to formally define formal and informal FD languages in a systematic and efficient way. The evaluation method is based on the set of formal criteria presented in Chapter 5. The comparison method mainly shows how to adapt and transform the informal or formal definitions of FD languages in order to render them suitable for comparison.

- **Chapter 8:** *Free Feature Diagram Language (FFD) : A Formal Configurable Definition* presents the definition of Free Feature Diagram language (FFD) following principles presented in Chapter 4. We mainly present and discuss the semantic proposed originally for VFD and then for FFD. More precisely this chapter describes how we have formalised the original semantics of FODA FD that was previously given in natural language. The FFD definition is original in the sense that it is configurable. Accordingly, FFD enables to formally redefine most of the FD languages introduced in Chapter 6. This definition is reused in Chapter 9 to provide a formal definition to informal FD languages and in Chapter 10 to implement the VFD semantics within a reasoning tool.

- **Chapter 9:** *Feature Diagram Languages: Quality Analysis* presents the quality evaluation and comparison of (1) the informal FD languages covered by FFD (Chapter 8), (2) three formal FD languages: our FD language (VFD) (Bontemps et al., 2004; Schobbens et al., 2007), van Deursen *et al.*'s FD language (vDFD) (van Deursen and Klint, 2002) and Batory's FD language (BFT) (Batory, 2005) and (3) Boolean Circuits (BC) (Shannon, 1937, 1938; Vollmer, 1999). Although BCs are not a FD language, our comparison method remains applicable. FDs are closely related to Boolean logic. Hence, one could question whether visual languages associated to Boolean logic may not replace FDs advantageously. To answer this question we propose to apply our method to compare FFD and BC.

  These quality analyses are based on the criteria defined in Chapter 5 and on the method presented in Chapter 7.

Part IV, Automation of Feature Diagram Languages presents solutions to support automation and reasoning on VFD.

- **Chapter 10:** *Varied Feature Diagram Language (VFD): A Reasoning Tool* describes a tool based on the semantics defined in Chapter 8 and automating various decision problems related to FD languages.

Part V, Conclusion and Future Work summarises the major findings and discusses possible future work.

- **Chapter 11:** *Conclusion and Future Work* first presents conclusions related to the research problem and questions, then recalls the claimed contributions and their limitations. Finally, suggestions for future work are provided.

Part I

Research Domain

Chapter 1

Software Product Lines

In this chapter, we recall the main concepts of Software Product Lines (SPL) and underline why variability is a major issue in SPL Engineering (SPLE), specifically during Requirements Engineering. Subsequently, we discuss why variability modelling is crucial and why we address it. This will lead us to the second chapter that describes Feature Modelling and how it contributes to variability modelling within SPL.

The structure of this chapter is as follows. Firstly, we introduce SPL in Section 1.1. The main points that we address are: SPL process and principles (Section 1.1.2), SPL evolution and adaptation (Section 1.1.3), SPL benefits (Section 1.1.4), SPL challenges (Section 1.1.5) and concrete applications successfully applying SPL principles (Section 1.1.6).

Secondly, in Section 1.2 we underline that variability is a crucial issue in SPL. Variability is defined in Section 1.2.1 and its main activities are described in Section 1.2.2. Categorisation and levels of variability are respectively addressed in Section 1.2.3 and 1.2.4. Several variability mechanisms are presented in Section 1.2.5. Finally, variability modelling is discussed in Section 1.2.6.

1.1 Software Product Lines

1.1.1 Introduction

In software engineering, software reuse (Jacobson et al., 1997) has been a popular topic since the publication of the seminal papers: *Mass Produced Software Components* (McIlroy, 1968) and *On the Design and Development of Program Families* (Parnas, 1976). Nowadays, many researchers and organisations still believe in software reuse as a powerful means to improve productivity and quality in software development. A general definition of "software reuse" is given in (Krueger, 1992): "Software reuse is using existing artifacts during the construction of a new software system".

More recently, a new approach has been proposed to improve reusability within software development. This approach is called Software Product Lines (SPL) and leads to economies of scale, reduction in terms of cost and time to market, better quality and planning (Clements and Northrop, 2001). Previously, other proposals (Table 1) were aiming at similar objectives. However, the SPL

approach goes further and proposes to institutionalise systematic reuse throughout all software development phases. In more details, the SPL approach proposes solutions to cope with various problems that may occur during software development:

- Most software products in the same application domain have been built before at some levels. Industries, such as telecommunications, home appliances or car industry, build the same or similar products over and over again. Therefore, on specific market segments, software development should evolve from single product development to multiple products development based on massive reuse.

- Software producers interests are often in contradiction with customers interests. Indeed, producers want to maximise their benefits, minimising their production costs and time to market, while customers ask for better quality software tailored to their individual needs.

- Software products rapidly evolve and increase in size and complexity with high product diversity. Typical problems occur when product complexity increases:

    - The same functionalities are developed several times at different places,

    - The same changes are repeated at different places,

    - Identical features behave differently according to a particular product,

    - Updating products and managing changes are challenging tasks,

    - Maintaining products requires too much efforts and resources.

A *SPL* is defined as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* (Clements and Northrop, 2001).

SPL is a new step forward in the field of reusability and represents an innovative and growing approach in software engineering. Reuse is achieved through the development of a set of core assets shared among all members of the SPL. *Core assets* are common development artifacts such as requirement's documents, conception's diagrams, architectures, codes (reusable components), test's procedures or maintenance's procedures. SPL offers a new strategy, for designers and programmers, to cope with the emergence of new similar products. The main ideas behind this strategy are:

1. New products are not implemented from scratch but the implementation is driven by a planned, proactive and systematic reuse of the core assets,

2. A specific development process prescribes how the core assets should be built and how they should be reused when developing final products,

3. New necessary artifacts might be developed,

4. Various reuse mechanisms might be used such as parametrisation, inheritance, plugins, etc.

1.1.2 Software Product Line Engineering

Software Product Line Engineering (SPLE) is "a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation" (Pohl et al., 2005). The purpose is to reduce time and costs during development and to increase software quality and efficiency by reusing core assets that have been already tested and secured.

As illustrated in Figure 1.1, SPLE dedicates a specific process, named *Domain Engineering*, to the development of *core assets*. These core assets are then reused extensively during the second process called *Application Engineering* that corresponds to the development of the final products.
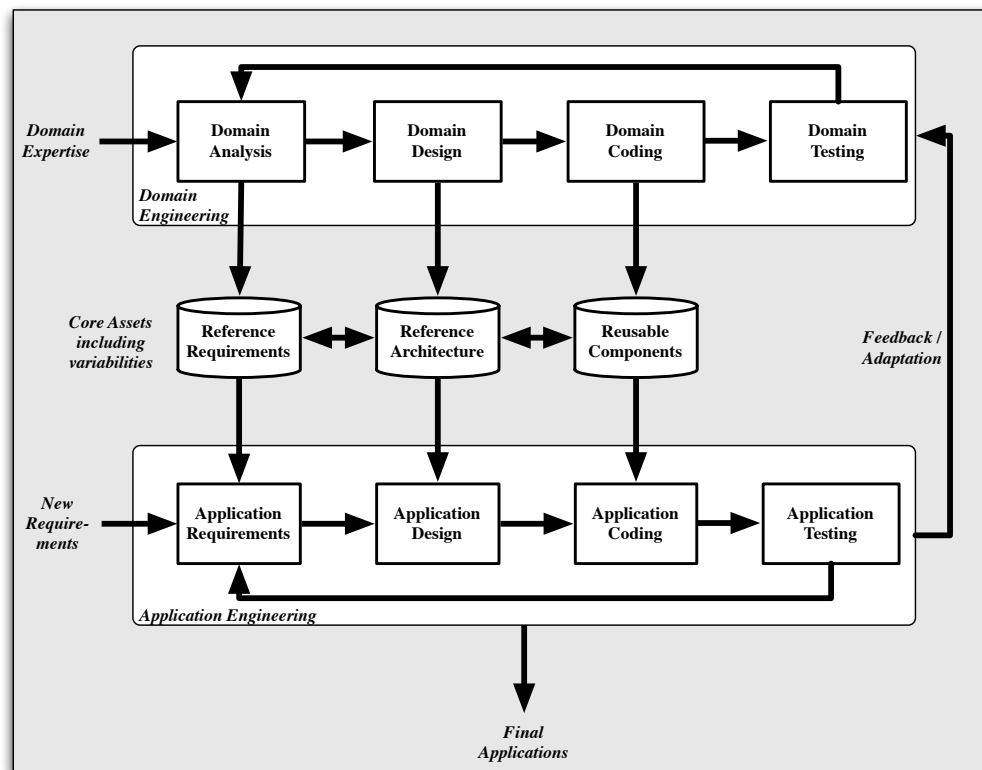
Figure 1.1: Software Product Line Engineering adapted from (Foreman, 1996; Pohl et al., 2005)

Both processes are related with traceability relationships to facilitate SPL evolution. **Backward and forward traceability** are established between reference requirements, reference architecture and reusable components to manage changes in SPL. A **feedback loop** is used during application engineering to revise domain analysis, design, coding and testing. New products developed within a SPL may contain specific requirements. These requirements could remain specific to a product or a decision could be taken to integrate them into the SPL. New products may reveal the necessity to modify reusable components or to integrate new reusable components into the SPL.

1.1.2.1   Domain Engineering

Domain Engineering, a.k.a development *for reuse*, is composed of four activities: domain analysis, domain design, domain coding and domain testing. Its main purposes are:

1. to identify which products will be included in the SPL,

2. to identify what differs between products, how it differs and where,

3. to identify and implement core assets.

In more details, Domain Engineering determines which functionalities are common to all products (a.k.a commonalities) and which differentiate one product from another (a.k.a variabilities). Once this distinction is made explicit, commonalities and variabilities are implemented. This process also determines where variabilities may occur, identifying variation points. Variation points are taken into account in the reference architecture where reusable components are selected and correctly combined to constitute the final product. Experience of the company is also often reused, e.g. coding standards, patterns and frameworks.

At the end, the outputs of this process are the scope of the family and its core assets. Core assets form the basis for the SPL and include, but are not limited to: reference requirements, reference architecture, reusable software components. Other development artifacts are often also included, such as domain models, documentation and specifications, performance models, schedules, budgets estimation, test plans, test cases, work plans, process descriptions,... (Clements and Northrop, 2008a). Here is a short description of these elements:

- The *scope of the family* is defined as the set of products included in the SPL. It often consists of (1) an enumerated list of product names, (2) a list of characteristics (functionalities) that the products have all in common, and (3) the ways in which they vary from one to another (Clements and Northrop, 2008a). This scope is a statement, made by the company, about which systems will be part of the product line and which will not. This set will obviously vary during time: market will evolve and variation in organisations' strategy will occur.

  A good definition of the scope is crucial, because if *too broad* it renders the core assets too complex to be effectively reused, and *too narrow* it does not justify the cost of core asset development and maintenance.

- The set of *reference requirements* gathers the commonalities and variabilities that will be reused to elicit requirements for a new product. Once the variabilities have been selected for the product, the designer may turn all his/her attention to the product particularities (a.k.a unexpected variabilities). These particularities are unforeseen requirements completely specific to a customer. First, they must be elicited and then integrated in the final product.

- The *reference architecture* defines (1) the infrastructure common to all the final products and (2) the interfaces of the reusable components that will be extensively reused during the development of the final products.

- *Reusable components* are software components reused during final product coding.

- *Production plans* specify a prescribed way to derive final products from the core assets. It specifies:

  1. how the core assets will be reused to build a product,

  2. the SPL's constraints to which the products must conform,

  3. the links between core assets and products.

  In other words, a production plan corresponds to a developers' guide that helps reusing core assets in a correct and efficient manner.

1.1.2.2   Application Engineering

Application Engineering, a.k.a development *with reuse*, is composed of four activities: application requirements engineering, application design, application coding and application testing. This process is built on the previous one and consists in developing a final product, reusing the core assets and adapting the final product to specific requirements. Its inputs are the core assets and its output the final product. At this stage, reuse is driven by:

1. the *reference requirements* that determined which requirements will be reuse, and therefore, by following the traceability links, which reusable components,

2. the *reference architecture* and *production plans* that determine how reusable components can be combined.

On step further is to automate application engineering. This is the main proposal of Generative Programming (GP) (Eisenecker and Czarnecki, 2000) that seeks to replace manual search, adaptation, and assembly of components by the automatic generation of needed components given a particular requirements specification. More specific product derivation approaches and techniques have been also proposed. Typical examples of such approaches are: the component-based approach (Atkinson et al., 2001), the feature-oriented programming approach (Batory, 2003), the aspect-oriented programming approach (Lee et al., 2006) and the model-driven approach (Botterweck et al., 2007).

1.1.3   Software Product Line Adoption and Evolution

A SPL does not appear accidentally, but requires a conscious and explicit effort from the interested organisation (Bosch, 2002). Firstly, the organisation may adopt an *evolutionary* or a *revolutionary approach*. The company has the choice to create its SPL from scratch (maybe using the old system at the same time to avoid interrupting the production) or to evolve its current system incrementally. Secondly, the SPL approach can be applied to an *existing line of products* or to a *new system*.

Each case has associated risks and benefits that should not be neglected. In general, the revolutionary approach involves more risks, but higher returns compared to the evolutionary approach (Bosch, 2002). Nevertheless, we can rarely expect a company to realise a complete SPL life-cycle immediately from the start. Most would start small, one component at a time, within pilot projects.

Once the SPL approach has been adopted, mechanisms have been proposed to manage it and make it evolve. One proposal (Northrop, 2002) is to divide SPL evolution into three essential activities: the *core asset development*, the *product development* and the *SPL management*. These activities can not be considered sequentially. Products and core assets are influenced by each other and evolve in parallel during their life cycle. For instance, core assets are reused to produce new products, but they are also defined on the basis of preexisting products and in prevision of upcoming products. They also evolve with the appearance of new products.

1.1.4   Software Product Line Benefits

The SPL approach can generate various benefits classified according to: organisational benefits, software engineering benefits and business benefits (Clements and Northrop, 2001).

Organisational benefits gather advantages like: a better domain comprehension, better mobility and training for employees, high quality products and customer's trust increase.

Software engineering benefits include advantages like: better requirements and components reusability, better requirements analysis, better software quality control, programming standards establishment, means to erase redundant implementations and more complete and reusable documentation.

Last but not least, **business benefits** mainly concern the diminution of production, maintenance and test costs. In addition, SPL improves process efficiency, budget planning and time schedule. For a detailed description of SPL benefits we refer to section "benefits and costs of product line" in (Clements and Northrop, 2001).

Several figures and statistics have been provided in (Baas et al., 2003) to illustrate notably the improvements in costs, time to market and productivity:

- Nokia is able to produce 25 to 30 different phone models per year (up from 4 per year) because of the SPL approach.

- Cummins, Inc., was able to reduce the time it takes to produce software for a diesel engine from about a year to about a week.

- Motorola observed a 400% productivity improvement in a family of one-way pagers.

- Hewlett-Packard reported a time to market reduced by a factor of seven and a productivity increase by a factor of six, for a family of printer systems.

Other success stories for SPL are gathered in the hall of fame for product lines (Clements and Northrop, 2008b) maintained by the SEI.

1.1.5 Software Product Line Challenges

Applying SPL principles necessitates important investments, generates risks and impacts both organisation and management. In more details, the company may be confronted to the following challenging problems (Clements and Northrop, 2001):

1. **Estimate economic results**. The initial investment is generally consequent and the first products to be developed within a SPL are often delayed. Real costs are difficult to estimate and therefore the return on investment is difficult to plan. Despite the potential future benefits, these problems are crucial especially for small and medium companies. When we compare traditional (Figure 1) and SPL development (Figure 1.1), we can underline that the SPL approach needs a long term management, because the visible benefits are not immediate. Indeed, as illustrated in Figure 1.2, the adoption of a SPL approach requires a considerable initial investment. In addition, the break even point would, generally, be reached at middle term, delaying the return on investment. A sufficient number of products must be developed to earn the real benefits.
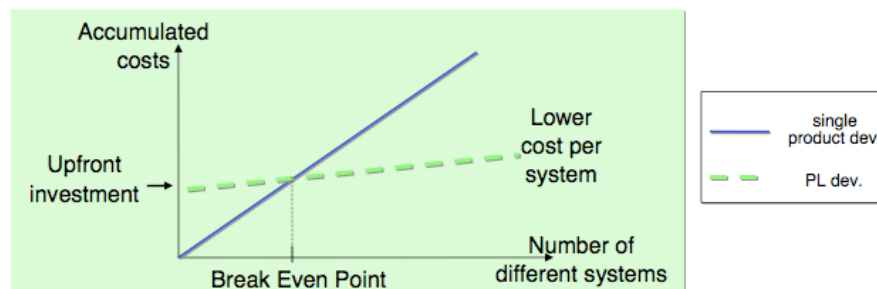
Figure 1.2: SPL Break Even Point (Pohl et al., 2005, p.10)

2. **Resistance to changes**. Changing the organisation's original development scheme from a single product to a product line approach entails a fundamental shift in mentalities and in the organisation itself. Such a shift should be carefully managed. Everyone in the organisation should feel involved and be convinced by the benefits of the approach for the organisation and for himself.

3. **Difficulty to introduce new technologies**. Introducing a new process is very risky when no development process has been previously defined explicitly and used in the company. Under project the new process will be quickly ignored and previous habits reestablished (Knauber et al., 2000).

4. **Evolution and flexibility**. Evolution is a complex problem. Evolution within a SPL is even harder as the SPL must remain consistent, even when two final products evolve in opposite directions. Some difficulties may appear when a new product should be included within the SPL. It requires to fix cautiously the SPL scope. In addition, once a core asset is shared between several products its evolution should be carefully controlled.

Problems	Percent responding
Organisational resistance	52 %
Management resistance	36%
Developer resistance	32%
Concerns over large investment	45%
Lack of properly trained staff	29%
Inability to measure impact	19%
Concerns of long lead time	18%

Table 1.1: Software Product Line Risks (Cohen, 2002)

To be efficient, a SPL is often less flexible. Indeed, the application domain should remain stable and the strategy for the past, present and upcoming products should remain coherent.

Once the SPL needs to evolve, the relevant information should be efficiently communicated through the organisation. The SPL and the way it evolves must be known by everyone in time. Although it allows rapid exchange of information, oral communication is not sufficient. Better communication mechanisms should be used to guarantee information accessibility and non-alternation.

5. **Difficulty to find a champion**. A champion is a person who has enough experience, responsibility and authority inside the company. He is familiar with the SPL principles and possesses a general overview over (1) the company's application domain and (2) its past, present and future products. This kind of profile is rare, especially in small and medium companies. Indeed, most of the time, nobody corresponds and when one corresponds, his or her workload is too big to take in charge such responsibilities.

Thus SPL adoption is mainly hindered by organisational risks. In (Cohen, 2002), different problems (see Table 1.1) are listed underlining that organisational and management resistance are the most critical.

1.1.6   Software Product Line Applications

SPL principles are applied to many kinds of software-intensive systems: telecommunications (Maccari, 2001; Bosch, 2005), home appliances (Kolberg et al., 2003), car industry (Thiel et al., 2001), etc.

In the following, we shortly describe five representative and successful applications of the SPL approach. Our goal is to underline that proactive and systematic reuse is applicable to many different systems in many different domains and organisations. More details on these success stories and others can be found in (Clements and Northrop, 2008b; Pohl et al., 2005).

1.1.6.1 Car Periphery Supervision (CPS)

Car periphery supervision systems (Thiel et al., 2001) are a family of automotive products designed to guarantee more safety and comfort for car drivers. These systems are based on sensors installed around the vehicle to monitor its local environment.

The functionalities provided by such systems are parking assistance, automatic detection of objects in vehicles blind spots and the adaptive control of airbags and seat-belt tensioners.

- **Commonalities** mainly concern data analysis. Indeed, each car periphery supervision system typically requests, filters, and evaluates data from sensors while at the same time performing diagnostics, availability tests, and consistency checks.

- **Variabilities** mainly concern car models, sensor measurement methods and data evaluation mechanisms.

1.1.6.2 Celsius Tech: Ship System 2000

Ship system 2000 (Brownsword and Clements, 1996) is a family of naval defence systems unifying all weapon systems, command-and-control, and communication functions on a warship.

- **Commonalities** mainly concern communication, fire-control and electronic warfare.

- **Variabilities** mainly concern ship types (surface or submarine) and classes, weapon systems, languages (English, French, etc.) and platforms.

1.1.6.3 Nokia: Mobile phones

Mobile phone systems are family of systems offering telecommunication services described in (Maccari, 2001; Bosch, 2005). For example, the functionalities provided by those systems are: voice communication, messages, clock, alarm clock, countdown timer, organiser, calculator, currency converter, reminders, games, ...

- **Commonalities** mainly concern basic communication functionalities (call, dial, handle gsm protocol, handle sms protocol, etc.), contact groups, message boxes, ...

- **Variabilities** mainly concern imaging, connectivity, number of keys, display sizes, languages, communication protocols and games.

1.1.6.4 Smart Homes

Smart home systems (Kolberg et al., 2003) are a family of systems providing extensive automatic control of home appliances. This includes services that basically enable to control heating, ventilation, security and energy. Those can be automatically or remotely controlled through a network interface connecting the home appliance to the Internet. In the end, fridges, televisions, stoves, lights, curtains, security cameras, stereos and ambient temperatures can be controlled.

- **Commonalities** mainly concern automatic control of appliances, sensor systems and security protocols.

- **Variabilities** mainly concern the level and type of services proposed according to the inhabitants and the type of building equipped with these systems. For instance adequate services can be provided for elderly people living in houses equipped with health control devices.

1.1.6.5 Weather Stations

Weather station systems (Kuusela and Savolainen, 2000) are a family of systems measuring weather-related phenomena in the environment and transmitting this information.

- **Commonalities** mainly concern sensor systems and algorithms to convert sensor data into meaningful information, such as wind speed and direction. Moreover, weather station rules and constraints are highly standardised.

- **Variabilities** mainly concern ways of communicating and information formats (digital or analog). Furthermore, the environmental conditions (polar regions, equatorial regions, dry regions) impose specific requirements on the system design. Indeed, many different kinds of weather stations exist such as ice weather stations, agricultural weather stations, high tower/NRC weather stations, water resource weather stations and shipboard weather stations.

1.2   Variability

One of the main challenges to obtain systematic reuse within SPL is variability management. Variability must be anticipated and documented to facilitate SPLE. Unfortunately, variability cannot always be anticipated and may introduce conflicts or unexpected interactions. Variability shows how products may vary within a SPL. In addition, SPL evolves over time when new products are engaged in and when existing products are evolving. These evolutions impact variability overview and may gradually degrade the SPL at all development phases. Hence, variability plays a central role in the entire SPL development process from requirements elicitation to architecture, components, coding and tests.

1.2.1   Definition of Variability

Variability is defined as the "ability to change or customise a system" (van Gurp et al., 2001). Weiss and Lai define variability in SPL as "an assumption about how members of a family may differ from each other"(Weiss and Lai, 1999). *Variabilities* specify the particularities of a system corresponding to the specific expectations of a client. *Commonalities* specify assumptions that are true for each member of the family.

Variability is usually described with variation points and variants. A *variation point* locates the place where variability may occur. In other words, it identifies where a choice between specific variants should be made. Each *variant* corresponds to one possible alternative that could be selected to resolve the variation point.

1.2.2   Variability Activities

Handling variabilities is decomposed into four essential activities suggested in (van Gurp et al., 2001):

- *Variability identification* determines product differences (variants) and their location (variation point) within the SPL artifacts.

- *Variability delimitation* defines variation point binding times and multiplicities.

- *Variability implementation* determines implementation mechanisms.

- *Variant management* makes evolve variants and variation points.

1.2.3   Categories of Variability

Variability appears in every software development phase. During variability identification, various sources may be identified as the origin of variabilities. Bachmann and Bass propose in (Bachmann and Bass, 2001) to classify variabilities into categories where each category corresponds to one possible source of variation (Figure 1.3):

- **Variability in function** means that a particular function may exist in some products and not in others.

- **Variability in data** means that a particular data structure may vary from one product to another.

- **Variability in control flow** means that a particular pattern of interaction may vary from one product to another.

- **Variability in technology** means that from a technical point of view the platform (OS, hardware, dependency on middleware, user interface, run-time system for programming languages) may vary from one product to another.

- **Variability in product quality goals** means that goals like security, performance or adaptability may vary from one product to another.

- **Variability in product environment** means that the product domain may impose different specific requirements to any products.

Another classification, based on the distinction between essential and technical variability, has been proposed in (Halmans and Pohl, 2003). *Essential variability* refers to the customer's viewpoint who cares about usage aspects concerning functional and non-functional requirements. *Technical variability* refers to the product family engineer's viewpoint who cares about variation points, variants and binding times to realise variability. Essential variability defines what to implement and technical variability defines how to implement it. These two classes of variability have also been refined in various subcategories that are summarised in Figures 1.4 and 1.5. For more details we refer to (Halmans and Pohl, 2003).
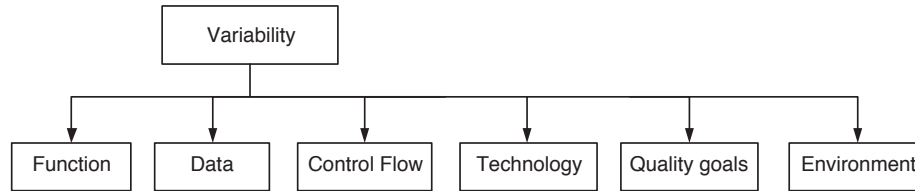
Figure 1.3: Classification of variability
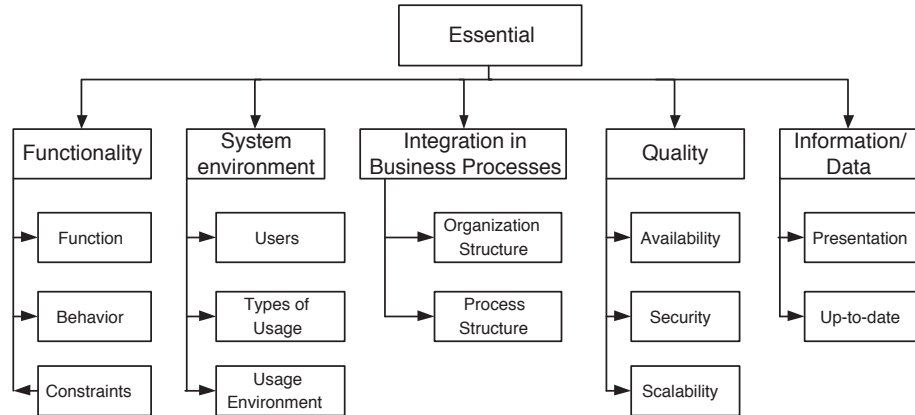
Figure 1.4: Essential product family variability (Halmans and Pohl, 2003)
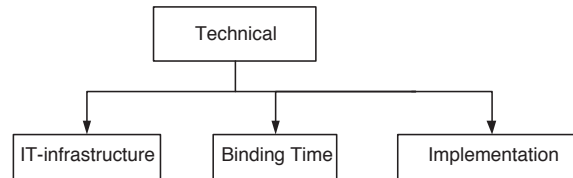
Figure 1.5: Technical product family variability (Halmans and Pohl, 2003)

Following the same approach, Becker *et al.* distinguish *Specification* and *Implementation variabilities* in their meta-model of variability (Becker, 2003).

Another categorisation has been proposed by (Metzger et al., 2007). The authors distinguish *Software* and *Product Line variabilities*.

- Software variability refers to the "ability of a software to be efficiently extended, changed, customised or configured for use in a particular context" (Svahnberg et al., 2002).

- Product Line variability refers to the "variation between the systems that belong to a Product Line in terms of properties and qualities, like features that are provided or requirements that are fulfilled"(Metzger et al., 2007).

Product Line variability adopts the business (market segmentation) point of view while software variability adopts the SPL engineer point of view. An important challenge is to guarantee a correspondence between business and technical constraints.

In the end, we identify four categories of variability corresponding to each concern:

1. one addresses the SPL *developer*'s point of view (Technical and Implementation variabilities),

2. one addresses the SPL *designer*'s point of view (Specification, Internal and Software variabilities),

3. one addresses the *customer*'s point of view (Essential and External variabilities),

4. one addresses the *business*' point of view (Product Line variability).

In the sequel, we refer to those categories respectively as *developer, designer, customer and business variabilities*.

1.2.4 Levels of Variability

Concerning the *designer variability*, Svahnberg, *et al.* have defined five levels of variability (Svahnberg and Bosch, 2000) where different variability design issues appear:

- **SPL Level**. Variability concerns how products differ, i.e, which components are used by different products and which product specific code (PSC) is used.

- **Product Level**. Variability concerns the architecture and choice of components for a particular product. On the product level, the components are fit together to form a product architecture, and the PSC is customised for the particular product variation. The variability issues on this level are:

  1. how to fit components together,
  2. how to cope with evolving interfaces, and
  3. how to extract and/or replace parts of the PSC.

- **Component Level**. This level is where the set of framework implementations are selected. The variability issues on this level are:

  1. how to enable addition and usage of several component implementations,
  2. how to design the component interfaces in such a way that it survives the addition of more concrete implementations.

- **Sub-Component Level**. A component consists of a number of feature sets. On the sub-component level the unnecessary features are removed to avoid dead code and others are added.

- **Code Level**. The code-level is where evolution and variability between products are implemented, ideally matching the provided and the required class interface defined in previous steps.

Variability Mechanism	Development phase	Description
Inheritance	At class definition time	Specialisation is done by modifying or adding to existing definitions.
Configuration	Previous to run-time	A separate resource, such as file, is used to specialise the component.
Parametrisation	At component implementation time	A functional definition is written in terms of unbound elements that are supplied when actual use is made of the definition.
Template instantiation	At component implementation time	A specification type is written in terms of unbound elements that are supplied when actual use is made of the specification.
Generation	Before or during run-time	A tool that produces definitions from user input.

Table 1.2: Variability Mechanisms adapted from (Jacobson et al., 1997)

1.2.5 Variability Mechanisms

Concerning the *developer variability*, several variability mechanisms exist. In (Jacobson et al., 1997), the authors identify various variability mechanisms (described in Table 1.2) and situate them according to the development phases where to apply them.

In (Svahnberg and Bosch, 2000), the authors make the links between developer and designer variability. They describe which variability mechanisms can be used at which variability levels. They underline the central role played by two variability mechanisms: configuration and parametrisation. Indeed, configuration is useful at higher variability levels (i.e, the SPL, product and component levels). Parametrisation is useful at lower levels (i.e, the component, sub-component and code levels). Other mechanisms such as inheritance and extension are also used at all levels of variability because they can be easily combined with other mechanisms.

Nevertheless, in industry, parametrisation is often used as the only mechanism for all variability levels using `ifdefs`, for instance. This is generally a bad habit that leads to difficulties in managing variability and to maintain an efficient and consistent SPL. This goes along the lines of Krueger who recognises that `ifdefs` are often inappropriate to implement variabilities as they violate the principle of separation of concerns:

> " `Ifdefs` *and related preprocessor approaches mix product line logic with runtime application logic. This mixing of concerns does not scale because source code becomes increasingly opaque as multiple preprocessor conditionals are added to a source file.*" (Krueger, 2001)

1.2.6 Variability Modelling

Variability modelling is becoming more and more crucial in SPL. Variability models help documenting variation points and their variants. They facilitate the identification and delimitation of variabilities (Section 1.2.2). Managing variability becomes a primary concern to mitigate complexity that may grow exponentially when variability increases. It is necessary to model variation points and their variants to express variability (Jacobson et al., 1997). Variability needs to be managed already during requirements engineering. In the initial phases of a SPL project, an efficient communication between SPL designers and customers about requirements and hence **variabilities** is a critical success factor.

One way to communicate efficiently on variability is to model it. Variability modelling languages are therefore used to produce variability models. Variability modelling languages can be graphical, textual or a mixture of both. In the SPL context, two main approaches are proposed to model variability: (1) single product languages that are extended with variability concepts (Section 1.2.6.1) and (2) variability languages that are specifically dedicated to model variability independently from any other languages (Section 1.2.6.2).

1.2.6.1 Extended Single Product Languages

Some languages originally dedicated to model single products have been extended to include concepts such as variants or variation points. For instance, UML diagrams (OMG, 2008) or Goal Modelling Languages (van Lamsweerde, 2001; Mylopoulos, 2006) have been respectively extended for this purpose. One shortcoming is the lack of uniformity regarding the variability representation. Each language is extended with its own extension mechanisms and variability constructs. No global coherence between variability modelling languages is obtained. Different proposals to extend single product languages have been published:

- **UML Use Case Diagrams**: (Gomaa and Shin, 2002), (Halmans and Pohl, 2003), (Bertolino et al., 2002), (John and Muthig, 2002) and (van der Maßen and Lichter, 2002).

- **UML Class Diagrams**: (Clauss, 2001), (Gomaa, 2001) and (Choi et al., 2006).

- **UML Sequence Diagrams**: (Ziadi et al., 2002, 2003).

- **Goal Modelling Languages**: (González-Baixauli et al., 2004) and (Liaskos et al., 2006).

We note that even within UML the mechanisms used to extend the different types of diagrams can be heterogeneous.

- Add new variability constructs to one specific UML diagram notation. (Halmans and Pohl, 2003)

- Use UML stereotypes or profiles to extend the UML notation (Gomaa, 2001; Atkinson et al., 2001; Ziadi et al., 2003).

- Use meta-modelling to extend UML diagrams separately and to describe variability dependencies between models (Gomaa and Shin, 2002; van der Maßen and Lichter, 2002; Gomaa and Shin, 2004).

The last mechanism is a first step to harmonise variability in UML. However, if other non UML languages such as Goal Modelling Languages (Liaskos et al., 2006) are used in the modelling process these mechanisms are limited and difficult to perceive for non UML modellers.

1.2.6.2   Variability Modelling Languages

As variability is a cross-cutting concern in many (different types of) models, some authors have proposed to dedicate one specific model to represent variability. The first proposal was published by Kang *et al.* back in 1990. The authors have suggested a Feature-Oriented Domain Analysis language called FODA (Kang et al., 1990). This language provides a concise and explicit way to:

- centralise and model variability in terms of features,

- model dependencies between features,

- help the engineer to derive final products from the SPL (i.e. making decision concerning variability and therefore configuring the SPL),

- facilitate the reuse and evolution of software components implementing these features.

Since Kang *et al.*'s initial proposal, many other "feature diagram" (FD) dialects have appeared in the literature: (Kang et al., 1998; Griss et al., 1998; Eisenecker and Czarnecki, 2000; Eriksson et al., 2005; Riebisch et al., 2002; Riebisch, 2003; van Gurp et al., 2001; van Deursen and Klint, 2002; Asikainen et al., 2004; Cechticky et al., 2004; Czarnecki et al., 2004, 2005c,b; Antkiewicz and Czarnecki, 2004; Wang et al., 2005b; Sun et al., 2005; Wang et al., 2005a; Batory, 2005; Asikainen et al., 2006; Janota and Kiniry, 2007).

Once a language is used to model variability, the next issue is to link this variability model with other models in which no information about variability is initially described. The challenge is now to manage variability across models and hence for all modelling languages used during all development phases. Various approaches have been proposed to deal with this issue:

- Gomaa and Shin (Gomaa and Shin, 2002) have proposed a meta-model to deal with variability within UML models. This approach is restricted to UML models and does not explicitly possess the concept of variation point. Elements in models are stereotyped as mandatory, optional or variant.

- Bachmann *et al.* (Bachmann et al., 2003) have proposed a meta-model for variability with variants, variation points and rationale.

- Becker *et al.* (Becker, 2003) have proposed a meta-model for variability distinguishing specification from implementation variabilities and static from dynamic variation points.

- Sinnema *et al.* (Sinnema et al., 2004) have proposed a framework for modelling variability in SPL called COVAMOF. This framework enables (1) to model and organise hierarchically variability, (2) to model dependencies between variabilities and (3) to model relations between dependencies.

- We have proposed in (Trigaux and Heymans, 2005) a definition of a "varied construct and link notation" (VCLN) including the concepts of variation point, unit of reuse, dispatching and reuse links. The purpose of this language is to uniformly manage variability across all models that can be abstracted to simple constructs and links between them.

- Pohl *et al.* (Pohl et al., 2005) have proposed an Orthogonal Variability Model (OVM) that contains a variability meta-model and its graphical representation to model variability independently from other modelling aspects.

- Czarnecki and Antkiewicz (Czarnecki and Antkiewicz, 2005) have suggested to keep FD languages as the main variability model and to superimpose features to other model elements. Model elements are therefore tagged with feature names. Once a feature is not selected in a valid configuration the corresponding model elements are successively eliminated.

1.3 Chapter Summary

In this chapter, we have presented the general SPL approach with its basic concepts, its process, its advantages and disadvantages. We have pinpointed variability as one crucial challenge that should be tackled to achieve systematic reuse. Finally, we have suggested that variability modelling and particularly Feature Modelling is one way to improve it. In the sequel of the thesis, our investigation is mainly limited to the early phases of the SPL approach and more specifically to variability modelling in SPL requirements.

In the following chapter, we will present Feature Modelling and its basic concepts. In Chapter 9, we will show how we evaluate Feature Diagram languages in order to improve variability management and thus systematic reuse within SPL.

Chapter 2

Feature Modelling: The Basic Concepts

I n the previous Chapter, we have familiarised the reader with Software Product Lines. Variability modelling has been presented as a key issue that should be addressed during the whole SPL process and particularly during domain analysis. In this chapter, we will present the basic concepts of one approach used to model variability. This approach is called Feature Modelling. *Feature Modelling* is defined as *"the activity of modelling the common and the variable properties of concepts and their dependencies and organising them into a coherent model referred to as feature model (Eisenecker and Czarnecki, 2000)"*.

Feature Modelling has been considered as "the greatest contribution of Domain Engineering to Software Engineering (Eisenecker and Czarnecki, 2000)".

The structure of this chapter is as follows. Firstly, in Section 2.1, we describe the notion of Feature Diagram (FD). We present FDs' purposes (Section 2.1.1), a FD example (Section 2.1.2) and symbols and constructs included into FDs (Section 2.1.3). In Section 2.1.4 we distinguish and discuss three essential notions:"Feature Diagrams", "Feature Models" and "Feature Languages". Secondly in Section 2.2, we discuss the fuzzy notion of "feature".

2.1   On the Notion of Feature Diagrams

A *Feature Diagram* (FD) is a directed acyclic graph (DAG) describing the hierarchical decomposition of features (called parents) into sub-features (called children) and the dependencies between them:

- Features are represented by nodes;

- Feature decomposition is represented by directed links between nodes. The origin of the link is the parent and the destination of the link is one of its children;

- Constraints between features are represented by textual constraints or links that crosscut the DAG.

A *Feature Tree* (FT) is a restricted FD where feature sharing is not allowed. A feature is said "shared" by two parents when it is the destination of two different feature decompositions.

A FD represents variability in a SPL and is used to determine which products (feature combinations) are included in the SPL. Although listing valid products contained in a SPL is a primary concern, FDs are also relevant in other situations.

2.1.1 Feature Diagrams' Purposes

FDs are an essential means of communication between domain and application engineers (see Section 1.1.2) as well as customers and other stakeholders such as: marketing representatives, managers, etc. In particular, FDs play an essential role during requirements engineering (Kang et al., 1990). FDs provide a concise and explicit way to:

- describe allowed feature combinations and variabilities between them,

- represent features dependencies,

- guide the selection of features during the derivation of a specific product,

- facilitate the reuse and the evolution of software components implementing these features.

2.1.2 Feature Diagram Example

In less than twenty years, mobile phones have gone from being rare and expensive pieces of equipment used by businesses to pervasive low-cost personal items. Rapid evolution and adaptation of mobile devices and their systems have been necessary to obtain this tremendous shift. One of the main challenges for the mobile phone industry was to maintain a short development life-cycle for new mobile phone models in order to remain competitive and to limit costs. Companies involved in this industry face harsh competition and therefore need to adapt quickly to satisfy the increasing demand.

Mobile phone systems are embedded systems that offer various features: voice communication, text messaging, phonebook, calendar, internet browsing, games, etc. An embedded system is characterised by its strong dependency on the device it controls and in which the system is completely encapsulated. Features from the device and features from the system are tightly bound together. In addition, seamless dependencies exist between logically separate features and bring many unexpected difficulties. Eliciting and modelling these feature dependencies are likely to be profitable.

In this context, FDs have been used in order to model feature dependencies, to shorten life-cycle development, to allow rapid adaptation and to improve reuse. In Figure 2.1, a simplified FD using the FODA FD language (Kang et al., 1990) is presented. It describes a Mobile Phone PL in terms of features. These features frequently correspond to the mobile phone capabilities available to the customer. The FD enables to guide the customer choice according to the defined variability and the corresponding mobile phone models. In order to avoid any confusion brought by the term "model" and to keep vocabulary uniform throughout the thesis, we will call them products as they are included in a PL. Products that are composed of features. Hence, mobile phone models are now called mobile phone products.

A FD should be read from the top to the bottom. In this specific case, the features form a tree in which they are decomposed into sub-features. In Figure 2.1, the top feature, *MobilePhone*, corresponds to the whole PL (i.e. the Mobile Phone PL). It is decomposed into four sub-features: *Dial*,
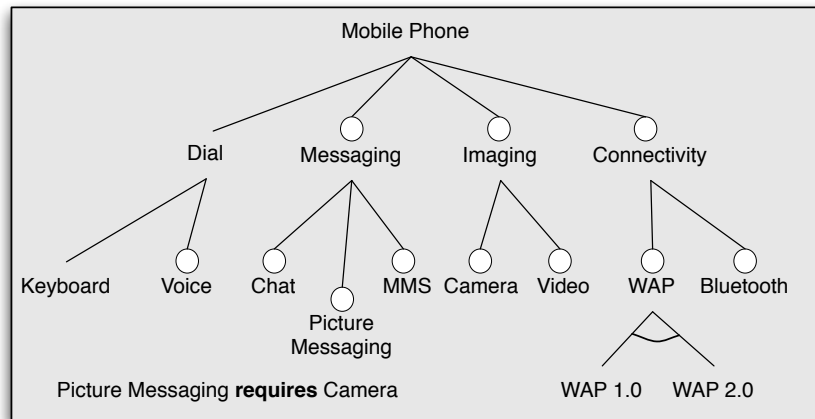
Figure 2.1: FODA FD: Mobile Phone PL

Messaging, *Imaging* and *Connectivity*. The feature *Dial* indicates that every Mobile Phone must be able to dial a phone number. This feature is also decomposed into two sub-features: *Keyboard* and *Voice*. This decomposition indicates that every product *must* be able to dial using keyboard and that some products *could* also allows dialling using voice recognition. The difference between "could" and "must" is represented graphically by the appearance of a hollow circle above the feature or not. *Keyboard* must be present as this feature is not decorated by a hollow circle while *Voice* could be present as this feature is decorated by it. It should be underlined that this graphical convention depends on the FD Language used, nevertheless the principle remains.

Many feature dependencies are expressed within a FD:

- Firstly, feature decompositions are dependencies and different *types of decompositions* exist. For instance the *WAP* feature is decomposed into sub-features *WAP* 1.0 and *WAP* 2.0 that mutually exclude each other. The feature *Imaging* is decomposed into sub-features *Camera* and *Video*. However, both features are optional (decorated with hollow circles), it means that either both, either only one, either none of them may be selected.

- Secondly, an implicit dependency between a sub-feature and its parent exist. Indeed, a sub-feature is only available for a product if at least one of its parents is selected for the product. Later, we will formalise this dependency as the *justification rule* (Definition 8.1.5, last point) where at least one parent is the justification of the sub-feature. In Figure 2.1, the feature *Dial* is the parent and justification of its son *Keyboard*. In our example, the FD is actually a FT and therefore each feature has exactly one parent. However, when FDs form DAGs, a sub-feature could be shared by several parents. The justification rule will be respected if at least one parent of the sub-feature is selected.

- Thirdly, explicit dependencies can be stated in a textual or graphical form. For instance,

the textual dependency *Picture Messaging* **requires** *Camera*, at the bottom of the figure, indicates that the feature *Camera* is needed by the feature *Picture Messaging*. It corresponds to a domain constraint stating that to send messages containing pictures, the mobile phone product should be equipped with a camera.

In the end, this FD and its constraints enables to represent a Mobile Phone PL containing 462 possible different products. Here are some of them, each represented as a feature set:

- {*Mobile Phone*, *Dial*, *Keyboard*}

- {*Mobile Phone*, *Dial*, *Keyboard*,*Voice*}

- {*Mobile Phone*, *Dial*, *Keyboard*,*Messaging*, *Picture Messaging*, *Imaging*, *Camera*}

- {*Mobile Phone*, *Dial*, *Keyboard*, *Messaging*, *MMS*, *Chat*, *Imaging*, *Camera*, *Video*, *Connectivity*, *WAP*, *WAP* 2.0, *Bluetooth*}

- etc.

2.1.3 Feature Diagram Constructs, Symbols & Meanings

In the last 15 years or so, research and industry have developed several FD languages with different constructs. To each of these constructs different (graphical) symbols and sometimes meanings have been associated. The first and seminal proposal was introduced as part of the FODA method back in 1990 (Kang et al., 1990). Since Kang *et al.*'s initial proposal, several extensions have been devised as part of the following methods: FORM (Kang et al., 1998), FeatureRSEB (Griss et al., 1998), Generative Programming (Eisenecker and Czarnecki, 2000), FORE (Riebisch et al., 2002; Riebisch, 2003), PLUSS (Eriksson et al., 2005), and in the work of the following authors: van Gurp *et al.* (van Gurp et al., 2001), van Deursen *et al.* (van Deursen and Klint, 2002), Czarnecki *et al.* (Czarnecki et al., 2005c,b), Batory (Batory, 2005), Benavides *et al.* (Benavides et al., 2005a) and Wang *et al.* (Wang et al., 2005a; Sun et al., 2005; Wang et al., 2005b).

All these languages will be presented in Chapter 6. In the next sections, we will first present the initial proposal by Kang and the basic principles shared by most proposals (see Section 2.1.3.1). Then, we will describe the principal extensions that have been provided throughout the years (see Section 2.1.3.2).

2.1.3.1 FODA FD Constructs, Symbols & Informal Meanings

As illustrated in Figure 2.2, a FODA FD is basically a tree with nodes related with various types of edges. A FODA FD is composed of various constructs to which symbols are associated:

- A **root**, a.k.a *concept*, is a feature that refers to the complete system. In Figure 2.2, the root or concept is the feature at the top of the tree, namely *MobilePhone*. The root derogates to the justification rule by being the only feature without any parent and that must be always selected.
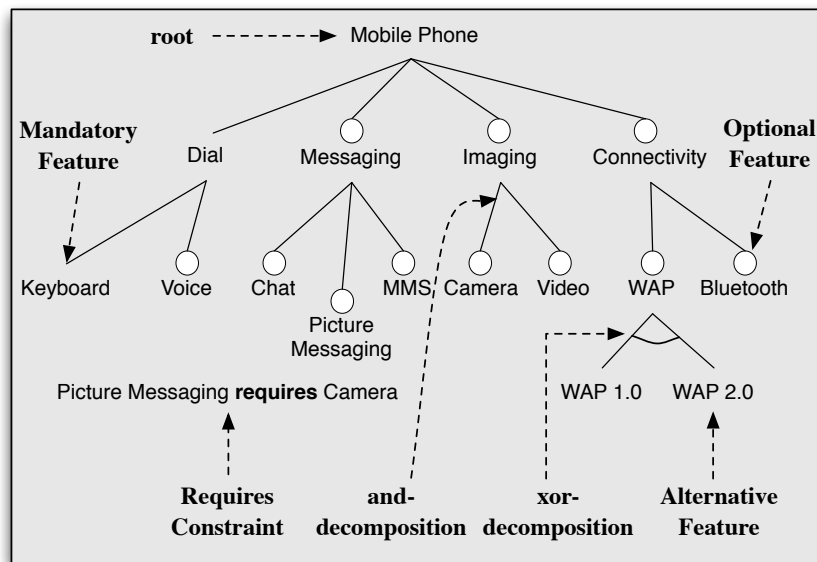
Figure 2.2: FODA FD: Constructs & Symbols

- *Features*, that can be *mandatory* (by default), *optional* or *alternative* and which are subject to decomposition.

  - **A mandatory feature** is part of a product iff at least one of its parents is in the product. In Figure 2.2, the feature *Keyboard* is mandatory.

  - An **optional feature** can be part of a product iff at least one of its parents is in the product. In Figure 2.2, the feature *Bluetooth* is optional.

  - An **alternative feature** or **xor-feature** is part of a product iff at least one of its parents is in the product and no other sub-feature of the selected parent(s) of this alternative feature is selected. In Figure 2.2, the features *WAP* 1.0 and *WAP* 2.0 are alternative and mutually exclude each other.

- *Dependencies* between features are materialised by:

  1. *Decomposition Edges* represented in a graphical form and that allow relating features that share at least one parent:

     (a) **and-decomposition** indicates that every sub-feature should be selected. In Figure 2.2, the feature *Imaging* is decomposed through an and-decomposition into features *Camera* and *Video*;

     (b) **xor-decomposition**, a.k.a alternative, indicates that only one sub-feature should be selected. In Figure 2.2, the feature *WAP* is decomposed through a xor-decomposition into features *WAP* 1.0 and *WAP* 2.0;

2. *Constraints* represented in a textual form and that allow relating features that do not share a parent:

   (a) **requires**, indicates that when a feature is selected, the other features that are required by it should be selected. In Figure 2.2, a requires constraint is expressed between features *Picture Messaging* and *Camera*;

   (b) **mutex**, an abbreviation for "mutually exclusive with", indicates that two features cannot be present simultaneously.

2.1.3.2   FD: More Constructs, Symbols & Meanings

Since Kang *et al.*'s initial proposal (Kang et al., 1990), several extensions have proposed new constructs for FD languages (see Figure 2.3):

- A new type of feature has been added: **or-feature** (Griss et al., 1998). An **or-feature** can be part of a product iff at least one of its parents is in the product. The additional constraint is that for each selected parent that is decomposed into or-feature(s) at least one of its or-features should be selected too;

- Three new types of decomposition edges have been added:

  1. **or-decomposition** (Griss et al., 1998) indicates that one or more sub-features should be selected;

  2. **card-decomposition** (Riebisch et al., 2002) indicates the minimum and maximum number of sub-features that should be selected;

  3. **group-decomposition** has been proposed by (Eisenecker and Czarnecki, 2000) and enables to decompose one feature with several types of feature decomposition. For instance, in Figure 2.3, the example illustrating the graphical representation of the construct group-decomposition indicates that the feature $f0$ is a feature on which two different types of decompositions apply: an and-decomposition where $f0$ is decomposed into $f3$ and $f4$ and a xor-decomposition where $f0$ is decomposed into $f1$ and $f2$;

- **Feature cardinalities** have been proposed by (Czarnecki et al., 2005b). According to their metamodel, a feature cardinality is an attribute of a leaf feature (non decomposed feature). This feature cardinality is an interval $< m, n >$ indicating the minimum (m) and maximum (n) number of times the sub-feature can be selected;

- **Group cardinalities** have been proposed by (Czarnecki et al., 2005b). A group cardinality defines an interval $< m, n >$ indicating a minimum (m) and maximum (n) number of distinct features that can be selected within a feature group. This construct is equivalent to **card-decomposition** proposed in (Riebisch et al., 2002);

- **FD references** have been proposed by (Czarnecki et al., 2002, 2005c). These references allow reuse and modularisation of FDs;

- **Feature binding times** have been proposed by (van Gurp et al., 2001). Feature binding time describes when a variable feature is to be bound, i.e. selected, to become a feature of the final product or not;

- **Feature attributes** have been proposed by (Griss et al., 1998) and later by (Benavides et al., 2005a) to express attribute-based constraints. Feature attributes are defined as a characteristic of a feature that can be measured (Benavides et al., 2005a);

- **New Types of dependencies** such as specialisation and implementation dependencies proposed in (Kang et al., 1998) or usage, modification and activation dependencies proposed in (Lee and Kang, 2004).

The main FD constructs are illustrated in Figure 2.3. They are associated to their original symbols and authors. The upper part of the figure presents the constructs of the FODA FD language (Kang et al., 1990) and the lower part presents some popular new constructs that appeared in various extensions of this language.

Variation in FD languages did not only come from new constructs, but also from new symbols being associated to previously existing constructs. For instance, graphical forms of *requires* and *mutex* constraints have been proposed in (Griss et al., 1998) (Figure 2.4). The proliferation of new symbols associated to the same constructs is confusing. Another more striking example concerns the five different graphical symbols proposed for the construct *xor-decomposition* (Figure 2.4).

Considering the meaning of these constructs, most proponents have not discussed it and refer implicitly to the informal semantics provided in FODA (Kang et al., 1990). However, when new constructs are introduced their semantics should be defined, and preferably formally. In this sense, some authors (van Deursen and Klint, 2002; Czarnecki et al., 2004, 2005b; Batory, 2005; Benavides et al., 2005a; Wang et al., 2005a; Sun et al., 2005; Wang et al., 2005b; Schobbens et al., 2007, 2006; Asikainen et al., 2006; Janota and Kiniry, 2007) have now proposed formal definitions of FD languages and their semantics.

2.1.4 The Distinction between Languages, Models and Diagrams

The distinction between languages, models and diagrams is not always clear. In this section we clarify this distinction and the terminology we use throughout the thesis in which FDs and FD languages play a central role.

2.1.4.1 Languages

According to (Harel and Rumpe, 2004, 2000), a *language* is a system of symbols used to represent and communicate information. A language is preferably defined by a concrete syntax, an abstract syntax and a semantics. A *concrete syntax* defines how the language elements appear in a concrete, human-usable form (symbols). An *abstract syntax* characterises in an abstract way the kinds of elements that make up the language and the rules to combine them. The *semantics* assigns an unambiguous meaning to each syntactically correct diagram written in this language. For example, if we consider a *FD language*, its definition should preferably be composed of:
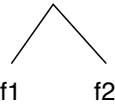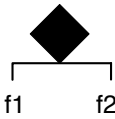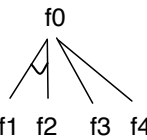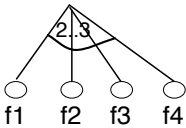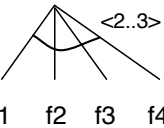
Construct	Symbol	
Optional Feature	f1	FODA Kang *et al.* 1990
And-decomposition	f1 f2	FODA Kang *et al.* 1990
Alternative Feature + Xor-decomposition	f1 f2	FODA Kang *et al.* 1990
Requires constraint	f1 **requires** f2	FODA Kang *et al.* 1990
Mutex constraint	f1 **mutex** f2	FODA Kang *et al.* 1990
Or Feature + Or-decomposition	f1 f2	Griss *et al.* 1998
Group-decomposition	f0 / f1 f2 f3 f4	Eisenecker and Czarnecki, 2000
Card-decomposition	2..3 / f1 f2 f3 f4	Riebisch *et al.* 2002
Group cardinality	<2..3> / f1 f2 f3 f4	Czarnecki *et al.* 2005b

Figure 2.3: FD: Constructs & Symbols

- a *Concrete Syntax* that defines the graphical symbols (see examples in Figure 2.3 column 2) that are used to draw FDs and that represent FD abstract constructs;
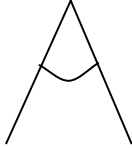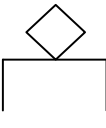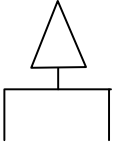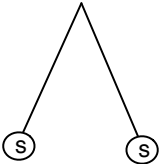
Construct	Symbol(s)
Requires constraint	f1 **requires** f2 f1 - - - - - - ➤ f2 FODA Griss *et al.* Kang *et al.* 1998 1990
Mutex constraint	f1 **mutex** f2 f1 ◄ - - - - - ➤ f2 FODA Griss *et al.* Kang *et al.* 1998 1990
Xor-decomposition	FODA, FORM Griss *et al.* van Gurp *et al.* Riebisch *et al.* Eriksson *et al.* Kang *et al.* 1998 2001 2002 2005 1990, 1998 Eisenecker and Czarnecki, 2000

Figure 2.4: FD: Constraints & xor-decomposition Symbols

- an *Abstract Syntax* that defines (1) the abstract constructs (or concepts) (see examples in Figure 2.3 column 1) used to build a FD and (2) the rules to combine them. For instance, one construct would be the *or-decomposition* (Griss et al., 1998) and one rule would be that the root has no parent;

- a *Semantics* that provides one unambiguous meaning to each FD respecting (1) the graphical conventions defined in the concrete syntax and (2) the rules defined in the abstract syntax. For instance, the informal semantics of an **or-decomposition** (Griss et al., 1998) is that at least one sub-feature of the parent must be selected.

2.1.4.2 Models

According to (Krogstie, 1995), a *model* is "an abstraction externalised in a language". A *model* is an abstract description focussing on some main aspects of the problem to solve. The main purpose is to prevent the analyst to be hampered by irrelevant aspects of the problem simultaneously. Therefore, a model is an abstraction that enables to separate concerns and isolates information on one specific aspect facilitating its management.

For example, a *Feature Model* is an abstract feature-based description of variability in a SPL. A Feature Model is usually composed of a graphical part, the FD, and of a textual part containing additional information over the FD. Examples of additional information are: feature descriptions, complex feature constraints, feature attributes, variability rationale, variability binding, variabil-

ity prioritisation, semantic descriptions, traceability links to requirements, code or other types of models, etc.

2.1.4.3   Diagrams

According to (Anderson, 1997), a *diagram* is a "pictorial, yet abstract, representations of information, and maps, line graphs, bar charts, engineering blueprints, and architects' sketches are all examples of diagrams, whereas photographs and video are not". A *Feature Diagram* is a DAG describing the hierarchical decomposition of features into sub-features and constraints between them. In the end, a FD is written in a FD language and is part of a feature model.

2.2   On the Notion of Feature

In the previous section, we have clarified the notions of diagrams, models and languages. Once this distinction has been made clear, one problematic question remains: What is exactly meant by feature?

Unfortunately, the term "feature" is heavily overloaded in the SPL community. Many different definitions have appeared including the following:

- A **feature** is "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems." (Kang et al., 1990)

- A **feature** is "a distinguishable characteristic of a system that is relevant to a stakeholder of the system." (Simos et al., 1996)

- A **feature** is "anything users or client programs might want to control about a concept." (Eisenecker and Czarnecki, 2000)

- A **feature** is "a logical unit of behaviour specified by a set of functional and non-functional requirements." (Bosch, 2000)

- A **feature** is "a capability or value which the user is willing to pay for." (Maccari, 2001)

- A **feature** is "a chunk of functionality that adds value to the product." (Maccari and Heie, 2003)

- A **feature** "describes a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements." (Chen et al., 2005)

- A **feature** "represents a distinguishable characteristic of a concept. A concept consists of a set of related features with constraints." (Sun et al., 2005);

- A **feature** is "an elaboration or augmentation of an entity(s) that introduces a new service, capability or relationship." (Batory, 2006)

- A **feature** is "an increment in product functionality." (Batory et al., 2006)

- A **feature** is "a property of your products that are meaningful to your customers and marketing people." (Pure-systems, 2007)

- A **feature** is "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option." (Apel et al., 2007)

Obtaining a consensus on the notion of feature seems unfeasible. The situation is very similar to the problems encountered when defining the notion of "requirement" in Requirements Engineering. Research communities have not reached a common agreement on the definition of both concepts.

The first attempt to categorise features has been proposed in (Kang et al., 1990) with the notion of layers, graphically represented in (Kang et al., 1998). Features are thus categorised according to the capability layer, operating environment layer, domain technology layer or implementation technique layer.

More recently, as presented in (Classen et al., 2007), the problem frames approach (Jackson, 1995, 2001) can help clarifying the concept of features by putting them in their context and disambiguating features that are requirements, domain properties or specifications. This approach allows a detailed and systematic analysis of variability according to a well established framework. Additionally, it allows automated reasoning and feature interaction detection. Still, one could argue that features are intended to be more general and may correspond to sets of requirements or mixture of requirements, domain properties and specifications (Classen et al., 2008).

Even though, we do not want to go much further into this debate. For this work, we simply avoid this problem by considering features as "black boxes". This hypothesis has no impact on our formal approach where features define indivisible elements of reasoning.

Nonetheless, as already introduced in Sections 2.1.3.1 and 2.1.3.2, a feature is characterised by a type and can be annotated with attributes.

2.2.1 Feature Types

Various types of features exist and mainly depends on how the parent(s) of the feature are decomposed. Classical feature types are:

- A **Mandatory feature** (Kang et al., 1990) is part of a product iff at least one of its parents is in the product.

- An **Optional feature** (Kang et al., 1990) can be part of a product when at least one of its parents is in the product.

- An **Alternative feature** (Kang et al., 1990) or **Xor-feature** is part of a product iff at least one of its parents is in the product and iff no other sub-features of the selected parent(s) of this alternative feature is selected.

- An **Or-feature** (Griss et al., 1998) can be part of a product iff at least one of its parents is in the product. The additional constraint is that for each selected parent that is decomposed into Or-feature(s) at least one of its Or-features should be selected too.

Another proposal (Eisenecker and Czarnecki, 2000) suggests to distinguish common and variable features. A **common feature** is a feature that is part of every product included in the SPL. Common feature should no be confused with mandatory features since the latter ones are not necessarily common to all products. On the contrary, **variable features** are all the features except the common features. By definition, *Optional*, *Alternative*, *Or*-features and their sub-features are variable features.

Later, other types of features have been provided:

- In (van Gurp et al., 2001) the authors define **external features**. An external feature refers to a feature offered by the platform of the system.

- In (Metzger et al., 2007) the authors make the distinction between **required** and **provided features**. A provided feature refers to a feature that the SPL offers. A required feature refers to a feature that the customer needs.

2.2.2 Feature Attributes

Feature attributes are defined as any characteristic of a feature that can be measured (Benavides et al., 2005a). Constraints on the attribute values may be specified. For instance, a value attribute may be associated to the feature "Horsepower" (see Figure 2). Additionally, a constraint may be defined for this attribute such has "Horsepower" should exceed 100 when "Air Conditioning" is selected ("Air Conditioning requires (Horsepower >100)"). Feature attributes are also used to document features with relevant information (Pohl et al., 2005) such as:

- *Feature Rationale* specifies **why** a feature is selected.

- *Feature Binding* specifies **when, where** and **how** a feature is selected.

    - Binding Time:
      The feature Binding Time specifies **when** a feature is selected. Typical binding times are: run-time, design-time, compilation-time or installation-time.

    - Binding Location:
      The feature Binding Location specifies **where** a feature is selected.

    - Binding Mode:
      The feature Binding Mode specifies **how** a feature is selected. Typically, two different modes are allowed: statically or dynamically bound.

- *Feature priorities*:
  Feature priorities specify how important a feature is. These priorities determine the feature implementation schedule according to the relevance of the features and/or their marketing advantages. However, these priorities may enter in conflict with the constraints described in the FD. For instance, if "$f1$ **requires** $f2$" and $f1$ has a higher priority than $f2$ then the priority of $f2$ should be at least as important as the one of $f1$.

2.3 Chapter Summary

Throughout this chapter, we have described the basic concepts of Feature Modelling. Our intention was essentially to familiarise the reader with Feature Diagram (FD) languages. Following a mobile phone example, we have presented their constructs, symbols and meanings. We have also discussed the notion of feature for which no consensus exists. We have provided informal descriptions of constructs appearing in specific FD languages. However, this list of constructs is far from being exhaustive since there is a profusion of FD languages. Even, the same relatively simple construct may be represented with different symbols and/or may have different meanings. The informality and the multiplicity of FD languages, symbols and meanings bring much confusion.

Throughout the rest of the thesis, we will try to reduce this confusion by improving the formalisation of FD languages, studying their quality and comparing them. In the next Part 2, we present the general concepts and theories on which we rely on to solve the aforementioned issues. First, we introduce, refine and situate ourselves within a generic quality framework notably designed to compare the quality of languages (Chapter 3). Then, we describe and adapt denotational semantic principles (Harel and Rumpe, 2004) to formally define modelling languages (see Chapter 4). Finally, we discuss and redefine formal criteria (see Chapter 5) that will be used later to compare FD languages (see Chapter 9) .

Part II

Quality of Models and Languages

Chapter 3

A Quality Framework

I n the previous chapter, we have presented the basic concepts of Feature Modelling. Throughout this chapter, we will present and refine a framework that will be further used to evaluate the quality of feature models and languages. The framework we use is the semiotic quality framework (SEQUAL) (Krogstie, 2001b; Krogstie et al., 2006). Our purpose is to refine it in order to evaluate languages according to well-known formal properties. Here, these formal properties are discussed and justified according to SEQUAL. They will then be formally defined in Chapter 5 and used in Chapter 9 to evaluate FD languages.

Modelling is a crucial activity in many fields such as economics, industrial and software engineering. A model is considered as an abstract representation of the domain. A model provides a representation of a domain that is understandable. Models constitute a means of communication to facilitate the dialogue between participants sharing different knowledge and views on the domain. A central problem is how to evaluate the quality of those models and languages used to produce them. Assessing and improving the quality of models and languages is a complex and multidimensional endeavour. To facilitate it, we suggest to use and refine the SEQUAL framework proposed by Krogstie *et al.* (Krogstie, 2001b,a; Krogstie et al., 2006). SEQUAL is arguably the most complete framework we know of. Its main advantages are:

1. to be neutral with respect to a particular type of models and languages,

2. to help situate quality analysis within a comprehensive quality space,

3. to act as a checklist of qualities to be pursued and

4. to recommend general guidelines on how the analysis should proceed.

However, SEQUAL does not detail on how to carry out specific quality evaluation or improvement tasks.

The structure of this chapter is as follows. Firstly, SEQUAL is briefly presented in Section 3.1. Its main concepts are introduced in Section 3.1.1. We describe, according to SEQUAL, how model and language qualities are evaluated respectively in Section 3.1.2 and Section 3.1.3. Then, this

framework is refined with formal language properties in Section 3.2. Finally, we briefly present some related work in Section 3.3.

3.1 SEQUAL Framework

The SEQUAL or semiotic quality framework has been developed over the last decade by Krogstie *et al.* (Krogstie, 2001b,a; Krogstie et al., 2006) as an extension of the original framework proposed in (Lindland et al., 1994). SEQUAL is grounded in semiotic theory that studies how humans communicate using signs. The theory is applied to modelling: elements in a model are considered as signs made to communicate meaning. SEQUAL is based on a distinction between various "semiotic levels": syntactic, semantic and pragmatic. It adheres to a constructivist world-view that recognises model creation as a part of a dialogue between participants whose knowledge changes as the process takes place. SEQUAL defines quality goals and sub-goals (called means) that contribute to achieve them. Different sub-goals were identified such as increasing the quality of modelling languages, processes, techniques or tools.

In (Krogstie, 2001b,a; Krogstie et al., 2006), one can find further details as well as suggestions of concrete means to pursue and measure the achievement of the quality goals. In the next three sections, we briefly present the concepts of the framework (Section 3.1.1), the quality goals for models (Section 3.1.2) and for languages (Section 3.1.3).

3.1.1 Framework Concepts

In (Krogstie, 2001b,a; Krogstie et al., 2006), Krogstie *et al.* have inventoried the elements involved in the modelling activity and used them to define quality for models and for languages. These elements are:

- M, the externalised model i.e., the set of all statements explicitly or implicitly made in the model.

- L, the language extension i.e., the set of all statements that are possible to make according to the graphical elements, vocabulary, and syntax of the modelling languages used.

- D, the domain i.e., the set of all statements that can be stated about the situation at hand. Enterprise domains are socially constructed and are more or less inter-subjectively agreed upon. That the world is socially constructed does not make it any less important to model that world.

- G, the organisationally motivated goals of the modelling task.

- K_S, the relevant explicit knowledge of the set of stakeholders (the audience A) involved in modelling . A subset of the audience is those actively involved in modelling with their knowledge indicated as K_M.

- I, the audience interpretation i.e., the set of all statements that the audience thinks an externalised model M consists of.

- *T*, the technical actor interpretation i.e., the statements in the model as interpreted by different model activators i.e., modelling tools.

3.1.2 Quality of Models (*M*)

In model quality, various general quality goals have been defined to study *M*. Models are evaluated and categorised according to them. These goals strongly depend on the models used and on the domain modelled. An overview of SEQUAL is given in Figure 3.1. It highlights various broad quality types that models should achieve.

- *Physical quality* pursues two basic goals: *Externalisation*, meaning that the explicit knowledge K_M of a participant has to be externalised in the model *M* by the use of a modelling language *L*; and *Internalisability*, meaning that the externalised model *M* can be made persistent and available, enabling other stakeholders to make sense of it. In other words, physical quality concerns the persistence and the availability of the model.

- *Empirical quality* pursues one goal: *Minimal error frequency*. It deals with error frequencies when different users read, write or encode model *M*, as well as ergonomy of human-computer interaction in modelling tools.

- *Syntactic quality* pursues one goal: *Syntactic correctness*. It deals with the correspondence between the model *M* and the language *L* in which *M* is written.

- *Semantic quality* pursues two goals: *Feasible completeness and validity*. It examines the correspondence between the model *M* and the domain of modelling, or universe of discourse, *D*. *Feasible completeness* examines that the model contains all statements about the domain that are relevant to the model except those that are not worthwhile trying to find. *Feasible validity* examines that all the statements made in the model are correct, except for the invalid ones that are not worthwhile trying to find.

- *Perceived semantic quality* pursues two goals: *Perceived completeness and validity*. It examines the correspondence between the audience interpretation *I* of a model *M* and the audience current explicit knowledge K_S.

- *Pragmatic quality* pursues one goal: *Feasible Comprehension*. It assesses the correspondence between the model *M* and its social and technical audience interpretations *I* and *T*, respectively. It evaluates to what extent people understand the models and to what extent tools can be developed to understand the models.

- *Social quality* pursues two goals: *Feasible Agreement and Resolution of conflict*. It seeks agreement among the audience interpretations *I*.

- *Organisational quality* pursues two goals: *Organisational completeness and validity*. It looks at how well the modelling goals *G* are fulfilled by the model *M*.
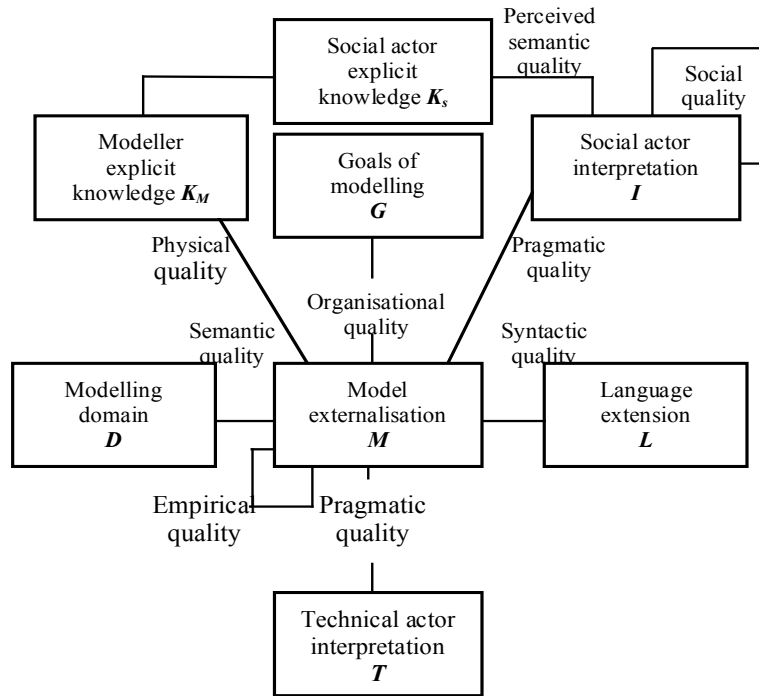
Figure 3.1: SEQUAL: Model Quality adapted from (Krogstie, 2001b; Krogstie et al., 2006)

3.1.3   Quality of Languages (*L*)

In (Krogstie, 2001b), language quality is assessed by the appropriateness of the modelling language *L* to achieve the model quality goals described above. SEQUAL has been adapted to evaluate language appropriateness (Figure 3.2). Six types of appropriateness and their corresponding goals were proposed, each of them concerns both the conceptual basis of the language (abstract syntax and semantics) and the visual representation of the language (concrete syntax).

- *Domain appropriateness* pursues *Domain suitability*. It deals with how suitable a language is for use within different domains. It means that language *L* must be powerful enough to express any statements in the domain *D*.

- *Participant language knowledge appropriateness* pursues *Language acquisition*. It measures if the audience knows the language or is able to easily learn it. It shows how the statements of *L* used by the audience match the explicit knowledge *K* of the audience.

- *Knowledge externalisability appropriateness* pursues *Semantic domain completeness*. It means that there are no statements in *K* that cannot be expressed in *L*.

- *Comprehensibility appropriateness* pursues *Concrete syntax suitability*. It measures that language users understand all possible statements of *L* and that these statements are closely

related to the way they think.

- *Technical actor interpretation appropriateness* pursues *Language automation.* It defines the degree to which the language lends itself to automatic reasoning and supports analysability and executability.

- *Organisational appropriateness* pursues *Organisational suitability.* It relates *L* to standards and other needs within the organisational context of modelling.
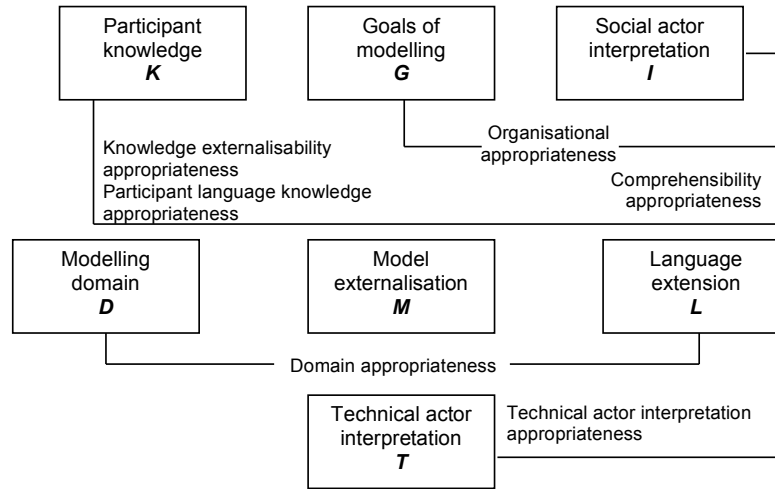
Figure 3.2: SEQUAL: Language Quality adapted from (Krogstie, 2001b; Krogstie et al., 2006)

3.2   Refining SEQUAL with Formal Language Properties

SEQUAL mainly targets two different concerns: quality of models and languages. However, using a high quality modelling language increases the chance of obtaining higher quality models. Hence, before starting any modelling activity, assessing modelling languages is decisive. Our purpose is to refine SEQUAL with tangible quality criteria that allow a formal quality evaluation of languages. These quality criteria are based on well-known formal language properties inspired from:

- The principles proposed in (Harel and Rumpe, 2004), that set the base to formally define modelling languages and their properties (see Chapter 4).

- Computational complexity theory (van Leeuwen, 1990; Papadimitriou, 1994), that allows studying the *complexity* of the decision problems needed to support the language.

- Languages theory (Hopcroft et al., 2000), that provides well-accepted formal language properties such as language *expressiveness*, *embeddability* and *succinctness* (see Chapter 5).

These formal criteria are defined and detailed in Chapter 4 and 5. In Table 3.1, we describe them briefly and informally.

Criteria	Informal Description
Formal Definition	A language is formally defined when its *abstract syntax*, its *semantic domain* and its *semantic function* are mathematically defined (Harel and Rumpe, 2004);
Complexity	Determines the potential efficiency of the algorithms to handle the language (van Leeuwen, 1990; Papadimitriou, 1994);
Expressiveness	Determines what the language is able to express (Hopcroft et al., 2000);
Embeddability	Determines whether the models written in one language are translatable into another language with translations (embeddings) that preserve the structure and the semantics of the models (Hopcroft et al., 2000);
Succinctness	Determines the worst impact on the size of the models when there is a translation from one language to another (Hopcroft et al., 2000).

Table 3.1: Formal Criteria: Informal Description

In Figure 3.3, we gather the different quality goals for models and languages and we show how the formal properties introduced above may contribute to them. Firstly, the reader should keep in mind that we do not aim at comparing languages quantitatively. We do not associate quantitative measures to all criteria. Furthermore, there are many factors that should be taken into account and these formal properties constitute only some of them. It is one possible refinement of SEQUAL and it should be complemented with other quality criteria. Secondly, the fulfilment of the *language* quality goals does not mean that the *model* quality goals are equally fulfilled. Indeed, good models can be obtained with poor modelling languages and conversely poor models can be obtained with comparatively good modelling languages.

The formal language properties have an impact on both *language* and *model* quality goals, directly or indirectly. In Section 3.2.1, we underline the contributions of these properties to language quality goals. In Section 3.2.2, the contributions of these properties to model quality goals are studied. Finally, in Section 3.2.3, we describe how quality of languages may contribute to quality of models.

3.2.1   Formal Criteria & Quality of Languages

When a language is formally defined, it possesses three main characteristics: an abstract syntax, a semantic domain and a semantic function (see Chapter 4). The abstract syntax should determine (1) the concepts used by the participants and (2) the rules followed by the participants to combine these
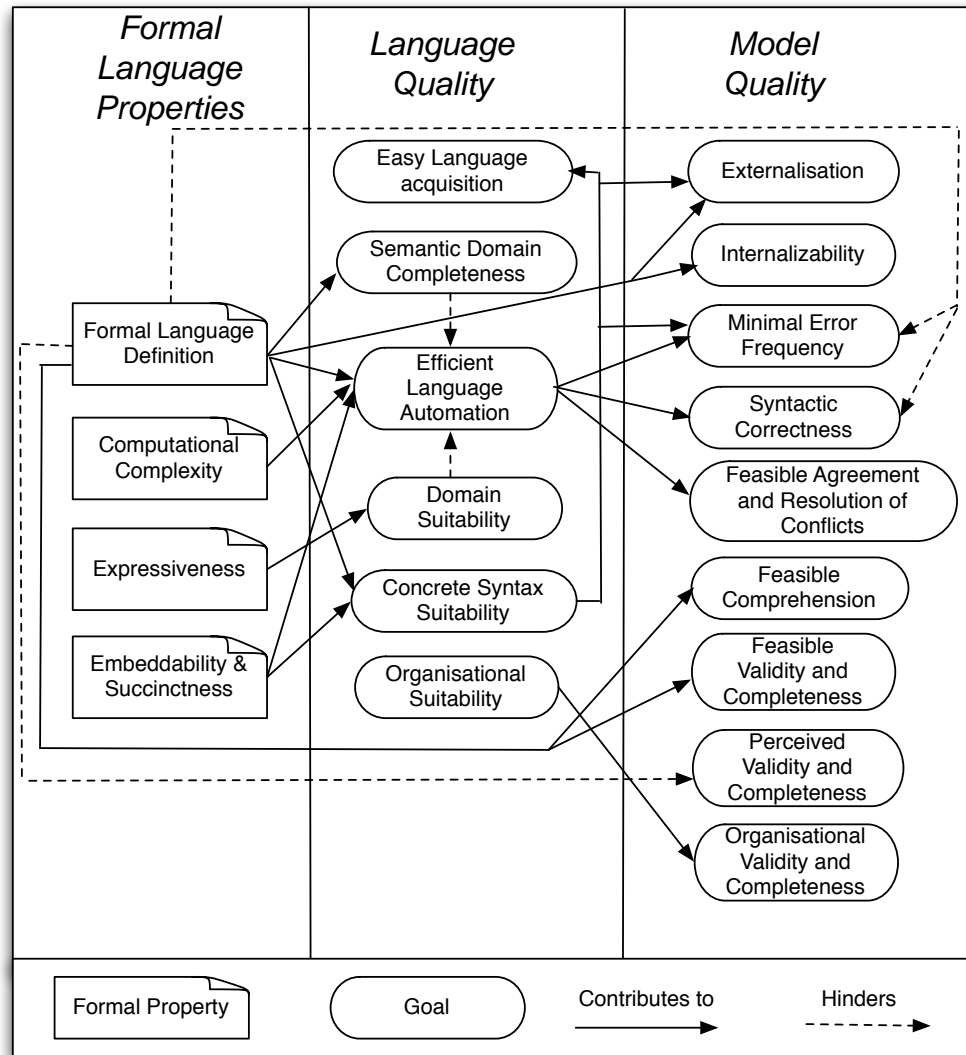
Figure 3.3: SEQUAL: Language Formal properties

concepts. The concrete syntax provides a graphical representation for each concept. Therefore, the abstract syntax contributes to *Concrete Syntax Suitability* (Figure 3.3) and *Comprehensibility Approriateness*. Indeed, *Concrete syntax suitability* does not only concern the graphical representation of the concepts but also how they are related to the way participants think.

The semantic function's definition assigns an unambiguous meaning to each syntactically correct model. Therefore, this function contributes to *Efficient language automation* (Figure 3.3) and *Technical Actor Interpretation Appropriateness*. Indeed, this function helps to define and implement decision problems associated to the language and its models. For instance, an interesting decision

problem is satisfiability: It checks whether a model has a non empty semantics or not. Satisfiability is formally defined as: given a model m and a semantic function $[\![\,]\!]$, is $[\![m]\!] \neq \emptyset$ true? One possible implementation would be: if $[\![m]\!] \neq \emptyset$ then *true* else *false*.

Although, most languages do not have a clear abstract syntax and semantics, they are amenable to them. Once their abstract syntax and semantics have been formally defined they can be compared according to four criteria (Table 3.1):

- The *Computational complexity* associated to the relevant decision problems that allow reasoning on the language is an indicator of the tool support scalability. Indeed, it studies from a theoretical perspective the potential efficiency or inefficiency of the decision problems implemented in the tool. Hence, *Computational complexity* contributes to *Efficient language automation* (Figure 3.3) and *Technical Actor Interpretation Appropriateness*.

- The *Expressiveness* of a language addresses *Domain Suitability* (Figure 3.3) and *Domain Appropriateness*. Indeed, the expressiveness studies whether every element of the semantic domain can result from the application of the semantic function to at least one model that conforms to the constraints of the abstract syntax.

- The *Embeddability* and *Succinctness* respectively study whether it exists a translation from one language to another that preserves structure (embeddability) and, if it exists, how this translation affects the size of the resulting models (succinctness). Hence, *Embeddability* and *Succinctness* contribute to *Efficient Language Automation* (Figure 3.3) and *Technical Actor Interpretation Appropriateness*. In addition, they contribute to *Concrete Syntax Suitability* (Figure 3.3) and *Comprehensibility Approriateness*. For instance, we assume that the concrete syntax of a language is not suitable when a language is translatable within itself (self-embeddable). Indeed, a language user might be confused when two different statements in one language share the same understanding. In addition, we assume that a language is more suitable than another when it is more succinct than the other.

We indicate in Table 3.2 which language quality dimensions (a.k.a. language appropriateness in (Krogstie, 2001a)) from the SEQUAL framework may be partially evaluated according to our selection of criteria.

Formal Criteria	**Domain appropriateness**	**Comprehensibility appropriateness**	**Technical actor interpretation appropriateness**
Formal Definition		√	√
Computational Complexity			√
Expressiveness	√		
Embeddability		√	√
Succinctness		√	√

Table 3.2: Formal Criteria and Language Appropriateness

3.2.2 Formal Criteria & Quality of Models

Both informal and formal language definitions may contribute to increase the modeller's explicit knowledge (K_M). Once the language definition becomes more familiar to the modellers, models are written (externalised) with more caution. The modeller is influenced by the constraints imposed by the abstract syntax and by the semantics that provides a model understanding that may differ from its intuitive understanding. Therefore, as soon as more constraints are defined, the frequency of errors is higher and the correspondence between the model and the language is more difficult to reach. From this point of view, *Formal Semantic Domain*, *Formal Semantic Function* and *Formal Abstract Syntax* hinders *Minimal Error Frequency* and *Syntactic Correctness*.

In addition, once the explicit knowledge of the audience (K_S) increases, the correspondence between the interpretation of the audience and its explicit knowledge may be more difficult to reach. For instance, with formal languages one has an unambiguous definition where no interpretation is possible while for informal languages there are many potential interpretations. Once one is not limited by a formal definition, every model can be understood differently and it becomes easier to align it with the audience interpretation. Hence, a formal language definition hinders *Perceived Validity and Completeness* since the perception of the audience is different.

Beside the positive contributions defined in Section 3.2.1, a formal language definition also contributes to *Feasible Comprehension* and *Feasible Validity and Completeness*. Indeed, when defining the language semantics, the main purpose is to find the best correspondence between the model and the intuitive audience understanding for this model (*Feasible Comprehension*). Moreover, the semantic function defines the correspondence between the formal abstract syntax and the semantic domain (*Feasible Validity and Completeness*). The *Formal Abstract Syntax* corresponds to the model formalisation and the *semantic domain* corresponds to the formalisation of the modelling domain.

In addition, a formal abstract syntax of language facilitates the *Externalisation* of the knowledge into a model. The abstract syntax has been constructed in order to relate each of its construct to a concept in the domain. These concepts correspond to the way participants think and will be easily used during the modelling activity. A formal abstract syntax also contributes to *Internalisability*. The abstract syntax corresponds to the language meta-model and can be easily made persistent and available by the use of a repository.

3.2.3 Quality of Languages & Quality of Models

The various contributions (positive or negative) presented here are not exhaustive and intentionally refer to contributions provided by the defined formal language properties. According to these properties, mainly two language qualities may influence the quality of models. The first one is *Efficient Language Automation* that helps (1) to minimise *Error frequency*, (2) to improve the correspondence between the model and the language and (3) to oppose and integrate different models in order to resolve conflict. The second one is *Concrete syntax suitability* that facilitates the language acquisition and the externalisation of knowledge into models written in the specified language. Therefore, the negative contributions of a Formal Language Definition impacting *Minimal error frequency* and *Syntactic correctness* are mitigated with tool support that helps to check and validate these models.

3.3   Other Frameworks and Criteria

As presented in the previous sections, our purpose is to refine SEQUAL with quality criteria that evaluate languages and mainly their foundations (semantics and abstract syntax). SEQUAL is known as the most complete framework to evaluate models and languages (Genero et al., 2005). However, the quality goals described in SEQUAL are too general to be directly applicable. Hence, many proposals (listed and analysed in (Genero et al., 2005)) define quality criteria that may fit into SEQUAL. According to (Genero et al., 2005), many of them are focused on *model* quality criteria (van Griethuysen, 1982; Moody, 1991; Batini et al., 1992; Reingruber and Gregory, 1994; Lindland et al., 1994; Moody and Shanks, 1994; Boman et al., 1997; Moody, 1998; Schuette and Rotthowe, 1998; Genero et al., 2000; Moody, 2003; Moody and Shanks, 2003; Moody, 2005; Berenguer et al., 2005; Moody, 2006a) and few propose *language* quality criteria (Kim and March, 1995; Rossi and Brinkkemper, 1996; Prasse, 1998; Matulevičius et al., 2006; Djebbi and Salinesi, 2006; Opdahl and Berio, 2006) and modelling process criteria (Moody, 1998; Maier, 2001).

3.3.1   Prasse's Framework

One major reference for language evaluation is the framework proposed in (Prasse, 1998) that is specifically dedicated to the evaluation of object oriented modelling languages. It provides a list of "criteria of investigation" informally defined in Table 3.4). Both SEQUAL and Prasse's frameworks distinguish language abstract syntax and semantics from concrete syntax. In this latter framework the criteria are categorised according to three perspectives: User-Relevant, Model-Relevant and Economic Criteria (Table 3.3).

User-Relevant Criteria	Model-Relevant Criteria	Economic Criteria
Usability	Unambiguity	Reusability
Clarity	Consistency	Extensibility
Understandability	Formalisation	
Adequacy	Integration	
Verification		
Expressive Power		

Table 3.3: Criteria of Investigation adapted from (Prasse, 1998)

It is difficult to establish a clear correspondence between the criteria we have selected (Table 3.1) and Prasse's criteria (Table 3.4). We notice that we address only a subset of Prasse's criteria: verification, expressive power, unambiguity, consistency, formalisation and integration (Table 3.5). However, the automation of the language and its efficiency is not addressed by any of Prasse's criteria.

3.3.2   Djebbi's Framework

Another evaluation framework has been provided in (Djebbi and Salinesi, 2006) where the evaluation criteria (Table 3.6) are originally dedicated to variability modelling languages. Some criteria are applicable to all modelling languages and others, that are not described here (Variability Type,

Criteria	Informal Description
Usability	Determines if a language is usable for humans in opposition to machines;
Clarity	Determines if a model corresponds directly to recognition patterns of the user;
Understandability	Conceptual understandability determines if the language concepts are adequate for modelling. Notational understandability examines the notation concerning ergonomic aspects like the one studied in (Moody, 2005);
Adequacy	Determines whether the model corresponds to its problem domain. By definition a model is a simplified and idealised description of a problem domain;
Verification	Determines if a model is not error-prone according to a problem domain;
Expressive Power	Determines how far essential facts of the problem domain can be expressed in a clear and natural manner;
Unambiguity	Determines if the interpretation of the language concepts and of the combining rules are unambiguous according to its abstract syntax and semantics;
Consistency	Determines if the model is consistent according to the concepts and rules of the language (abstract syntax);
Formalisation	Determines if the semantics of the language and its concepts are defined exactly;
Integration	Determines if the language can be easily integrated with other languages, with other paradigms and with other development processes;
Reusability	Determines if the language with its concepts and rules can be reused (language reusability). Determines if the language contains concepts that enable to reuse or extend models such as generalisation or specialisation (model reusability);
Extensibility	Determines if the language can be extended following formal rules and preserving the structure of its meta-model.

Table 3.4: Prasse's Criteria: Informal Description adapted from (Prasse, 1998)

Formal Criteria	Prasse's Criteria
Formal Definition	Formalisation, Consistency, Unambiguity
Computational Complexity	Verification
Expressiveness	Expressive Power
Embeddability	Expressive Power
Succinctness	Integration

Table 3.5: Formal Criteria and Prasse's Criteria

Variability Documentation and Variability Dependencies), are specific to variability modelling. In addition, most of these criteria do not provide a quality criteria but rather prescribe concepts and rules to be included in the abstract syntax of a variability modelling language. It is specific in the sense that it concerns the usage of FDs in a particular company, for a given kind of project. However the notion of a "good" modelling language is relative to the context of use of the language. The priorities to be put on the expected qualities and criteria are very likely to be different from one company, or project, to another.

Criteria	Informal Description
Readability	Expresses the facility to visually apprehend the model;
Clearness	Examines graphic arrangement of the elements that make the model;
Simplicity	Determines if the model contains a minimal number of objects;
Expressiveness	Determines if the model represents obviously the user's needs and can be easily understood without additional explanation;
Evolution	Determines if the model can evolve overtime and integrate changes;
Minimality	Determines if each concept is represented once and only once in a language;
Adaptability	Determines if the language is flexible enough to fit each company specific needs;
Scalability	Determines if the language allows modelling of large-scale systems;
Support	Determines if the language is supported by a tool that automates handling of models.

Table 3.6: Djebbi's Criteria: Informal Description adapted from (Djebbi and Salinesi, 2006)

It is difficult to establish a clear correspondence between the criteria we have selected (Table 3.1) and Djebbi's criteria (Table 3.6). We notice that we address only a subset of Djebbi's

criteria: simplicity, minimality and support (Table 3.7). The reader should note that the definitions of *expressiveness* given in Tables 3.1 and 3.6 are incompatible. Djebbi's definition of *expressiveness* is closer to *understandability* as defined by Prasse (see Table 3.4).

Formal Criteria	**Djebi's Criteria**
Formal Definition	
Computational Complexity	Support
Expressiveness	
Embeddability	Minimality
Succinctness	Simplicity

Table 3.7: Formal Criteria and Djebbi's Criteria

3.4 Chapter Summary

Throughout this chapter, we have presented and refined the SEQUAL framework used to evaluate the quality of models and languages. Our purpose is to provide formal criteria to compare languages and to discuss them according to SEQUAL. These criteria have the advantages to be (1) language-independent, (2) user-independent, (3) non-subjective and (4) based on established theories such as language theory (Hopcroft et al., 2000) and computational complexity theory (van Leeuwen, 1990; Papadimitriou, 1994). These formal criteria are by no means to be considered complete. Other suitable criteria should be provided to evaluate languages and models in further research. Many criteria exist to evaluate languages. We have mentioned the ones proposed in (Prasse, 1998) and (Djebbi and Salinesi, 2006). These criteria are described informally and cover more quality dimensions. We believe that both formal and informal approaches are complementary and should be combined to depict a better quality panorama.

Our purpose in the next chapters is to provide a formal description of our language-independent and formal criteria. How a language should be formally defined is described in Chapter 4 while the studied language criteria (complexity, expressiveness, embeddability and succinctness) are formally defined and discussed in Chapter 5.

Chapter 4

Languages: Formal Definition

In Chapter 3, we have presented the SEQUAL framework in order to evaluate models and languages. We have shown how formal language properties may contribute to language and model quality. Throughout this chapter, we present and detail how to define a language formally denotational semantics (Stoy, 1977) recalled in (Harel and Rumpe, 2000, 2004).

In their illuminating papers "Meaningful modelling: What's the Semantics of 'Semantics'?" (Harel and Rumpe, 2004) and "Syntax, Semantics and All That Stuff. Part I: The Basic Stuff" (Harel and Rumpe, 2000), the authors recognise that:

> "Much confusion surrounds the proper definition of complex modelling languages [...]. At the root of the problem is insufficient regard for the crucial distinction between syntax and true semantics and a failure to adhere to the nature and the purpose of each." (Harel and Rumpe, 2004)

Although their main purpose is to improve the definition of complex modelling languages such as the Unified Modelling Language (UML) (OMG, 2008), we argue that, even if FDs are far less complex, they are also "victims" of similar "mistreatments". Moreover, we claim that the added value provided by formalisation largely compensates the efforts needed to formally define a FD language. The objective of this chapter is to recall the basic principles from (Harel and Rumpe, 2000, 2004) used to formally define a language. In the subsequent chapters, we will show how, based on these notions, we have devised an approach to (re)define, assess and compare FD languages.

The structure of this chapter is as follows. First we recall and illustrate the principles proposed by (Harel and Rumpe, 2000, 2004) in Section 4.1 where we detail the notions of *Syntax* (Section 4.1.1) and *Semantics* (Section 4.1.2). Then we recall in Section 4.2 the main misconceptions over semantic definitions as introduced by the same authors.

4.1   Formal Definition of Language

Formally, Harel and Rumpe make it clear that the unambiguous definition of a modelling language, be it textual or graphical, must consist of three equally necessary elements: a *syntactic domain* ($\mathcal{L}$), a *semantic domain* ($\mathcal{S}$) and a *semantic function* ($\mathcal{M}$). Furthermore, the authors argue that $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{M}$ must all be defined *formally* i.e. mathematically, in order to keep the risk of ambiguity at a

minimum. Throughout the thesis we will use the following convention: the syntactic domain of a language X will be noted $\mathcal{L}_X$, the semantic function of a language X will be noted $\mathcal{M}_X$ and semantic domain of a language X will be noted $\mathcal{S}_X$.

Figure 4.1 summarises Harel and Rumpe's approach. This figure illustrates the three necessary elements that are respectively two sets and one function between them. The set on the left is $\mathcal{L}$ and contains all the diagrams one can write in $\mathcal{L}$. The set on the right is $\mathcal{S}$ and contains all the possible meanings of every diagram one can write in $\mathcal{L}$. The total function that relates these sets is $\mathcal{M}$: it maps every diagram of $\mathcal{L}$ to a meaning in $\mathcal{S}$. Two diagrams may share the same meaning, as illustrated in Figure 4.1 where two diagrams in $\mathcal{L}$ (*herDiagram* and *hisDiagram*) are mapped to the same meaning in $\mathcal{S}$, that is $\mathcal{M}(herDiagram) = \mathcal{M}(hisDiagram)$. Indeed, two diagrams with different graphical representations may share the same meaning. In contrast, two different meanings may not be mapped by the same diagram. Since $\mathcal{M}$ is a total function no ambiguity is possible: one diagram always maps to exactly one meaning.
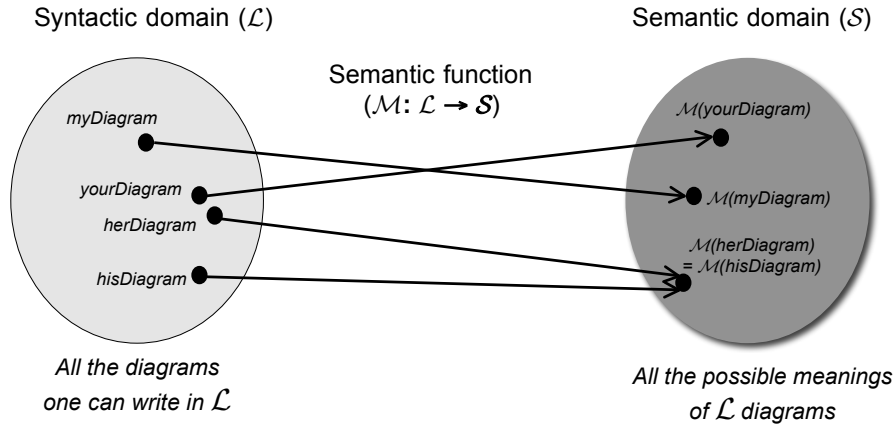
Figure 4.1: Formal language: the three constituents

A language with such formal $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{M}$ is called a *denotational formal language*. For *programming languages*, it is easy to understand why they need to be formal: given the same input, a given program must deliver exactly the same output whatever the interpreter or compiler that executes it. A formal programming language leaves no ambiguity in this regard to the implementers of interpreters. For *modelling languages*, we do not necessarily need to execute diagrams, especially if the language's purpose is not to represent behaviours that could be executed, animated or simulated in some way. However, most of the time, there is a great interest in carrying out precise computations on diagrams in order to derive properties about their meaning. For example, it might be extremely useful to have a tool that tells whether a given FD allows for at least one feature configuration, or if it is overconstrained and thereby allows none (Satisfiability, see Section 7.1). For a realistic FD, this verification is far from trivial and it can be very time-consuming and error-prone if left to humans. This type of verification is known as satisfiability checking and is one of the many FD-related tasks that can be automated (Benavides et al., 2006; Schobbens et al., 2007, 2006)

(see Section 7.1). If we want to describe languages in such a way that no ambiguity is left to tool developers, then we need a formal modelling language.

Finally, and maybe most importantly, we should not forget that modelling languages are mainly used to ease communication between human participants (Moody, 2006b). If a language has no well identified, clear and concise definition of a semantics to which one can refer in case of doubt, an expression in this language might well convey an unintended meaning. The semantic reference used to resolve conflicts may also evolve during the debates to reflect a consensus of its users. This is a necessary condition for the language and its tool support evolution and maintenance.

After this discussion on why a formal language definition is needed, we now give more details on how the syntax (Section 4.1.1) and the semantics (Section 4.1.2) of a language are defined according to denotational semantics (Harel and Rumpe, 2000, 2004).

4.1.1 Syntax Definition

In diagrammatic (a.k.a. visual, or graphical) languages, basic expressions include lines, arrows, closed curves, boxes and composition mechanisms involving connectivity, partitioning and "insideness" (Harel and Rumpe, 2004). These form the *physical representation* of the data (on screen, or on paper) that is known as *concrete syntax*. Most informal definitions of FD language semantics we found in the literature were based on concrete syntax, and usually discussed on FD examples. Most of the time, a substantial part of the semantics was implicit, leaving it to the diagrams to "speak for themselves". But actually, each readers' intuition is potentially different. And for computer support, we need to make everything explicit. As Harel and Rumpe put it:

> *"It is possible to guess the meaning of most terms, since a good language designer probably chooses keywords and special symbols with a meaning similar to some accepted norm. But a computer cannot act on such assumptions. To be useful in the computing arena, any language – whether it is textual or visual or used for programming, requirements, specification, or design – must come complete with rigid rules that clearly state allowable syntactic expressions and give rigid description of their meaning"* (Harel and Rumpe, 2004).

According to the state of the art in compilation and formal methods, it is better *not* to use concrete syntax as a basis for defining semantics. One reason is that, for visual languages, it appears particularly difficult to define rigid syntactic rules that clearly segregate between the allowed and the forbidden diagrams. Another reason is that, when based on the concrete syntax, the expression of the semantic interpretation rules is polluted by considerations related to visualisation, that makes the rules cumbersome. This is why the common practice in the aforementioned areas is to define the semantics of a language based on a so-called *abstract syntax*.

The *abstract syntax* is a representation of data that is independent of its physical representation and of the machine-internal structures and encodings. The set of all data that comply with a given abstract syntax is called the *syntactic domain*. Independence from machine-internal structures and encodings is useful (1) to simplify the description of the rules, and (2) to make the rules portable from one implementation to another.

For visual languages, the two most widespread ways to define an abstract syntax are: (1) *mathematical notation* (set theory) and (2) *meta-modelling*. In the latter case, the abstract syntax is

described through a so-called meta-model describing what is a well-formed (allowed) diagram. A meta-model is usually a UML Class Diagram (CD), possibly complemented with Object Constraint Language (OCL) rules (OMG, 2008). This format has the main advantage to be easily readable (UML CDs are a standard visual language with a partially well-accepted, although informal, semantics*), and to facilitate some tool implementation tasks, especially persistent storage of diagrams in a repository. Nevertheless, we prefer the mathematical format for its greater universality, unambiguity, conciseness and suitability to undergo rigorous proofs.

4.1.2   Semantic Definition

Once we have established a rigid set of syntactic rules, the role of a *semantics* is to assign an unambiguous meaning to each syntactically correct diagram. Harel and Rumpe recognise that *"agreement on a language's meaning is partly a sociological process, without which the communicated data is worthless"* (Harel and Rumpe, 2004). As we have seen in Chapter 3, this point of view is acknowledged and further elaborated in (Krogstie, 2001b; Krogstie et al., 2006) that adopts a constructivist view of the modelling activity. The sociological aspects of semantics are however out of the scope of this thesis. At the moment, we stick to the view where *"a language's semantics must provide the meaning of each expression, and that meaning must be an element in some well-defined and well-understood domain"* (Harel and Rumpe, 2004). Following (Harel and Rumpe, 2004, 2000), a semantics must have two main constituents: a *semantic domain* ($\mathcal{S}$) and a *semantic function* ($\mathcal{M}$). For describing them in the most universal, unambiguous and concise way, we opt again for mathematics.

4.1.2.1   Semantic Domain

According to denotational semantics (Harel and Rumpe, 2004), the first constituent of a semantics is the *semantic domain* ($\mathcal{S}$) that *"[...] specifies the very concepts that exist in the universe of discourse. As such, it serves as an abstraction of reality, capturing decisions about the kinds of things the language should express"*. Typically, a semantic domain is a mathematical domain built to have a structure that matches as much as possible to the real-world objects the language is used to account for, up to some level of fidelity. Having an explicit and well-defined semantic domain is crucial to get a clear idea of the kind of things that the modelling language is intended to represent. Without an explicit definition, it is difficult to judge the appropriateness of the language with respect to (1) the application domain, (2) the usage that the audience wants to make of it, and (3) the tools that will support it. Furthermore, looking at the semantic domain is necessary to compare two semantic definitions, as we will show in Chapter 7.

4.1.2.2   Semantic Function

The second constituent of a semantics is $\mathcal{M}$, the *semantic function*. It maps $\mathcal{L}$ to $\mathcal{S}$ and assigns a meaning to each syntactically allowable diagram (Figure 4.1). The *signature* of this function is therefore simply $\mathcal{M} : \mathcal{L} \rightarrow \mathcal{S}$.

---

*Several formal semantics of CD have been proposed in the literature (Kim and Carrington, 1999; Szlenk, 2006) but, at the time being, none appears in the official standard (OMG, 2008), nor as a de facto standard.

An important point is that the definition of $\mathcal{M}$ should be rigid too, that is, it should make it crystal clear which object (meaning) in $\mathcal{S}$ is assigned to any allowable diagram of $\mathcal{L}$. Since $\mathcal{M}$ is a *function*, there is one and only one such object for a given allowable diagram. Ambiguity in this context is therefore not possible.

Finally, the semantic function should be *total*, that is, it should not be possible to have a diagram in $\mathcal{L}$ which is not given a meaning in $\mathcal{S}$ by $\mathcal{M}$. A total semantic function ensures that the definition of the semantics is complete. The converse question will help language engineers to evaluate the *expressiveness* of their language (Section 5.2): is every element in $\mathcal{S}$ expressible by a diagram in $\mathcal{L}$?

4.2 Languages: Formal Definition and Misconceptions

The above may seem all too obvious to some readers. However, Harel and Rumpe have observed that many languages were never defined properly despite the benefits. Moreover, the authors have identified reasons why most of these languages are not properly defined. They have pointed out a set of frequent mistakes or misconceptions associated to semantics across software and systems engineering (Harel and Rumpe, 2004, 2000):

- **"Semantics is the meta-model"**. The most common mistake is to confuse the meta-model of the language and its semantics. The meta-model is one possible technique to describe the syntactic domain of the language, but not more. Usually, a meta-model is a combination of UML class diagrams (OMG, 2008) and OCL constraints. The class diagram specifies the allowed abstract syntax of the language and the OCL constraints specify additional rules but still syntactic rules that the diagram should respect. Nevertheless, nothing concerns the meaning of the diagram.

- **"Semantics is the semantic domain"**. The semantics could not be restricted to the semantic domain because the semantic domain is useless if no link is established between the representation of a diagram (an element in the syntactic domain) and its meaning (an element in the semantic domain). Hence, without a semantic function we have no semantics at all.

- **"Semantics is the context conditions"**. Context conditions entail neither a semantic domain nor a semantic mapping. They constrain the syntax as OCL constraints constrain the meta-model.

- **"Semantics is dealing with behaviour"**. Semantics is not restricted to behaviour definition. Some languages such as FD languages do not define behaviours, but they still need a semantics. Semantics may deal with behaviour but not always.

- **"Semantics is being executable"**. When a language is executable, it certainly has an operational semantics. However, even if the language is executable, there is no assurance that its execution is the intended one, that the semantics is adequately described and understood. In addition, the implementation aims at efficiency and deals with low level details while the semantic definition aims to highlight the underlying concepts, to be clear, precise but still understandable, and to facilitate equivalence proofs (do two diagrams have the same semantics?).

- **"Semantics is the meaning of individual construct(s)"**. The semantics of a language should be complete, all its constituent parts should be mathematically defined. However, defining the meaning of all its constituent parts is not always sufficient, a compositional semantics could be defined as a function on the meanings of all its constituent parts.

- **"Semantics means looking mathematical"**. Using mathematical symbols to define a language is often necessary but by no means a sufficient condition. The presence of mathematical symbols neither implies that there is a proper semantics, nor guarantees precision or unambiguity.

4.3   Chapter Summary

Throughout this chapter, we have presented how to formally define a modelling language according to (Harel and Rumpe, 2000, 2004). Various fundamental principles and concepts have been defined and discussed, such as *syntactic domain*, *semantic domain* and *semantic function*. They will be extensively reused throughout the next chapters. We have also underlined the main advantages associated with formal languages: unambiguity and language automation. In the following chapter (Chapter 5), we will reuse these principles and concepts to formally redefine language properties that will be used later to compare FD languages.

Chapter 5

Languages: Formal Criteria and Quality

Ⅰn Chapter 4, we have presented how Harel and Rumpe propose to formally define a language (Harel and Rumpe, 2000, 2004). Once languages are equipped with such a formal and denotational definition, we can compare them in a rigorous yet natural way. Our formal comparison is based on four criteria: *Complexity*, *Expressiveness*, *Embeddability* and *Succinctness*. Throughout this chapter, we define and discuss these criteria to evaluate the quality of languages.

The first criterion concerns the efficiency (computational complexity) of the algorithms used to reason on languages and to evaluate the scalability of the language according to defined decision problems. The three other selected criteria are commonly used in language theory (Hopcroft et al., 2000) and allows answering three questions:

- *Expressiveness*: What can the language express?

- *Embeddability*: When a diagram written in one language is translated into another language, does this translation preserve the structure of the diagram? If yes, we call this translation an embedding. The structure of a diagram usually corresponds to the shape formed by its nodes and edges.

- *Succinctness*: When a diagram written in one language is translated into another language, what's the impact of this translation on the size of the diagram? The size of a diagram is usually measured by the number of its nodes and edges.

These three criteria will be formally (re)defined in the next sections using Harel and Rumpe's concepts (Harel and Rumpe, 2000, 2004) already presented in Chapter 4.

The structure of this chapter is as follows. Each criteria is defined and discussed in turn. Computational *complexity* is defined in Section 5.1, *expressiveness* in Section 5.2, *embeddability* in Section 5.3 and finally *succinctness* in Section 5.4.

5.1 Complexity

Among its many advantages, formalising a language enables to rigorously define a set of questions (a.k.a decision problems) that can be asked about diagrams written in this language. These decision

problems concern the satisfaction of a certain "property" of the diagram. To automatically resolve them several algorithms can be provided and their efficiency should be carefully studied. Determining if a solution is efficient or not is only one facet of the problem. Indeed, one major problem is to identify whether the inefficiency is due to a suboptimal solution or to the difficulty of the problem in itself.

The theory that tackles these issues is complexity theory (Papadimitriou, 1994). In the context of languages, a *decision problem* is a question about a diagram that can be answered by 'yes' or 'no'. The efficiency of the algorithm that answers this question is related to how do the running time and memory requirements of the algorithm change given the size of the input.

For example, in formal languages that have a set as semantic domain, we can state precisely the *satisfiability* problem. The question we want to answer is: "is the given diagram satisfiable?". The diagram property we want to satisfy is satisfiability. A diagram is said satisfiable when its semantics is non empty. Satisfiability is naturally formalised as: given a semantic function $\mathcal{M}[\![\,]\!]$ and a diagram d, is $\mathcal{M}[\![d]\!] \neq \emptyset$ true?

Once these problems are formalised, one can ask (1) whether algorithms exist at all to solve this problem (*decidability*) and (2), if so, what is their optimal *complexity* in time and memory. A decision problem that can be solved by some algorithm is called decidable. A typical measure of complexity is *time complexity*, i.e., the number of steps for a Turing machine that it takes to solve an instance of the problem as a function of the size of the input, using the most efficient algorithm (Papadimitriou, 1994). Measuring the memory usage of the most efficient algorithm is called *space complexity*. Time and space are usually ranked into complexity classes (Papadimitriou, 1994):

- *P* is the set of decision problems solvable in polynomial time on a deterministic Turing machine.

- *NP* is the set of decision problems solvable in polynomial time on a non-deterministic Turing machine. This type of non-determinism is called "angelic": the Turing machine is supposed to make perfect guesses leading to the "yes" answer.

- A decision problem is *NP-complete* if it is in *NP* and every other problem in *NP* can be reduced to it in polynomial time. *NP-complete* problems are the most difficult problems in *NP* in the sense that they are the smallest subclass of *NP* that could conceivably remain outside of *P*. Classical *NP-complete* problems are: the Boolean satisfiability problem (SAT), the knapsack problem, the Hamiltonian cycle problem, the Travelling Salesman Problem (TSP), the graph colouring problem, ...

- *coNP* is the set of decision problems for which the complement is in *NP*, i.e. solvable in polynomial time on a non-deterministic Turing machine. The complement of a decision problem is obtained by swapping the yes and no answers. In other terms, *coNP* is the set of problems for which it is possible to find counter-examples in polynomial time on a non-deterministic Turing machine.

- The polynomial hierarchy is a hierarchy of complexity classes that generalize the classes *P*, *NP* and *coNP* to oracle machines. An oracle machine is a Turing machine connected to an

oracle that is able to resolve any problem of a given complexity class in a single step. In this case, a polynomial number of queries can be sent to the oracles. The class of problems solvable in polynomial time by a deterministic Turing machine calling an oracle solving problems in complexity C is noted $\mathbf{P^C}$, and similarly for $\mathbf{NP^C}$ and $\mathbf{coNP^C}$. Note that $\mathbf{P^P} = \mathbf{P}$, $\mathbf{NP^P} = \mathbf{NP}$ and $\mathbf{coNP^P} = \mathbf{coNP}$. For instance, the class of all languages decided by polynomial-time oracle machines calling a SAT oracle is noted $\mathbf{P^{SAT}}$ or $\mathbf{P^{NP}}$. According to (Papadimitriou, 1994, p. 424), the polynomial hierarchy is defined as the following sequence of classes:

- $\triangle_0 \mathbf{P} = \sum_0 \mathbf{P} = \prod_0 \mathbf{P} = \mathbf{P}$
- $\forall i \geq 0$:
  * $\triangle_{i+1} \mathbf{P} = \mathbf{P}^{\Sigma_i \mathbf{P}}$,
  * $\sum_{i+1} \mathbf{P} = \mathbf{NP}^{\Sigma_i \mathbf{P}}$,
  * $\prod_{i+1} \mathbf{P} = \mathbf{coNP}^{\Sigma_i \mathbf{P}}$.

Hence, $\prod_1 \mathbf{P} = \mathbf{coNP}$ and $\prod_2 \mathbf{P} = \mathbf{coNP^{NP}}$. In addition, if $\mathbf{P} = \mathbf{NP}$ the polynomial hierarchy collapses to the level 0, with $\mathbf{P} = \mathbf{NP} = \mathbf{coNP} = \triangle_i \mathbf{P} = \sum_i \mathbf{P} = \prod_i \mathbf{P}$.

- *PSPACE* is the set of decision problems that can be solved by a deterministic or non-deterministic Turing machine using a polynomial amount of memory (space) and unlimited time.

- A decision problem is *PSPACE-complete* if it is in *PSPACE* and every other problem in *PSPACE* can be reduced to it in polynomial time. One Classical *PSPACE-complete* problem is the Quantified Boolean Formula (QBF) problem (Papadimitriou, 1994, p. 455) or the satisfiability of linear temporal logic.

- EXPTIME is the set of decision problems solvable by a deterministic Turing machine in $O(2^{p(n)})$ time, where $p(n)$ is a polynomial function of n where n is the size of the input.

- NEXPTIME is the set of decision problems that can be solved by a non-deterministic Turing machine using time $O(2^{p(n)})$ for some polynomial function $p(n)$.

- EXPSPACE is the set of decision problems solvable by a Turing machine in $O(2^{p(n)})$ space, where $p(n)$ is a polynomial function of n.

In complexity theory, it is known that (Papadimitriou, 1994):
$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq NEXPTIME \subseteq EXPSPACE$$

Complexity results are important because they help evaluating (1) languages according to the complexity of their associated decision problems and (2) the scalability of tool support for this language. Formalisation of both the syntax and semantics is a necessary prerequisite to devise decision problems. Complexity analysis will then help identifying the worst case for each decision problem and to propose solutions to handle it. Heuristics taking into account the most usual cases can be added to the backbone algorithm, to obtain practical efficiency. Complexity results are good indicators to evaluate the quality of a language. When languages share common decision problems, their complexity with respect to these problems can be compared. Nevertheless, decision problems

may differ between languages, hence complexity results should be carefully analysed. For instance, a language X with a decision problem solvable by a deterministic Turing machine in $O(2^{p(n)})$ space (*EXPS PACE*) is not necessarily worse than a language Y with all its decision problem solvable in polynomial time on a deterministic Turing machine (*P*). *Y* is maybe insufficiently expressive to model real world problems (Section 5.2).

Obviously, other criteria are used to compare languages such as expressiveness, embeddability or succinctness. However, these criteria require that the languages share, or are amenable to, a common semantic domain. We start with expressiveness.

5.2 Expressiveness

The expressiveness of a language is commonly understood as what can be expressed in the language, more precisely in its syntactic domain. For a formal language, we can be more specific: the *expressiveness E* of a language *X*, noted *E(X)*, is the part of its semantic domain (S_X) that it can express, i.e., the *image* of its syntactic domain $(\mathcal{L}_X)$ through its semantic function $(\mathcal{M}_X)$. This is what Definition 5.2.1 says, and what Figure 5.1 illustrates. The syntactic domain of the language X $(\mathcal{L}_X)$ has an image $E(X) = \mathcal{M}_X(\mathcal{L}_X)$, a subset of *X*'s semantic domain (S_X).

Definition 5.2.1 (Expressiveness) *The* expressiveness *of a language X is the set $E(X) = \{\mathcal{M}_X[\![d]\!] | d \in \mathcal{L}_X\}$, also noted $\mathcal{M}_X[\![\mathcal{L}_X]\!]$. A language X is* more expressive *than a language Y if $E(X) \supset E(Y)$.*

If S is the common semantic domain of several languages, say W, X, Y and Z , their respective expressiveness can be compared. They form a partial order.

In Figure 5.1 we illustrate a situation where, because of their respective definitions, no two languages have the same expressiveness. For instance, *Z is more expressive than Y*: $E(Z) \supset E(Y)$. The expressiveness of *X* and *Y* are disjoint: $E(X) \bigcap E(Y) = \emptyset$. The expressiveness of *X* and *Z* overlap: $E(X) \bigcap E(Z) \neq \emptyset$ but are incomparable: $E(X) \not\subset E(Z)$ and $E(Z) \not\subset E(X)$. In general, the relationships between the syntactic domains (disjoint, overlapping, equal) of several languages are non-correlated with the relationships existing between their respective semantic domains. This is because the semantic function can be different for each language.

In Figure 5.1, we also notice that $E(W) = S$. In cases like this, when the image of $\mathcal{L}_W$ is the whole of S_W, we say that the language *W* is *expressively complete* (Definition 5.2.2), or in other words, the part of the semantic domain it can express is the semantic domain itself.

Definition 5.2.2 (Expressive Completeness) *A language W with a semantic domain S_W is* expressively complete *if $E(W) = S_W$.*

The usual way to prove that a language *Y* is at least as expressive as *X* is to provide a *translation* (Definition 5.2.3) between their syntactic domain from $\mathcal{L}_X$ to $\mathcal{L}_Y$.

Definition 5.2.3 (Translation) *A* translation *is a total function $\mathcal{T} : \mathcal{L}_X \rightarrow \mathcal{L}_Y$ that* preserves semantics*: $\mathcal{M}_Y[\![\mathcal{T}(d_1)]\!] = \mathcal{M}_X[\![d_1]\!]$.*
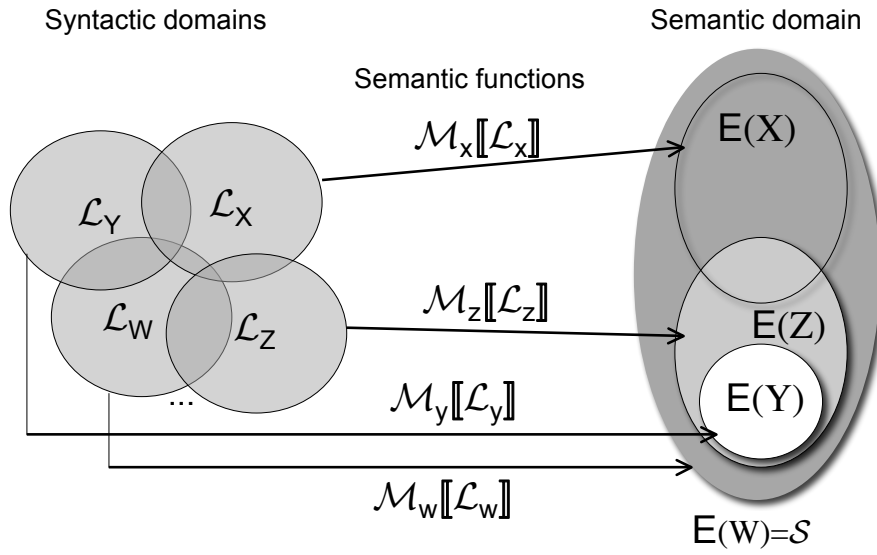
Figure 5.1: Comparing expressiveness
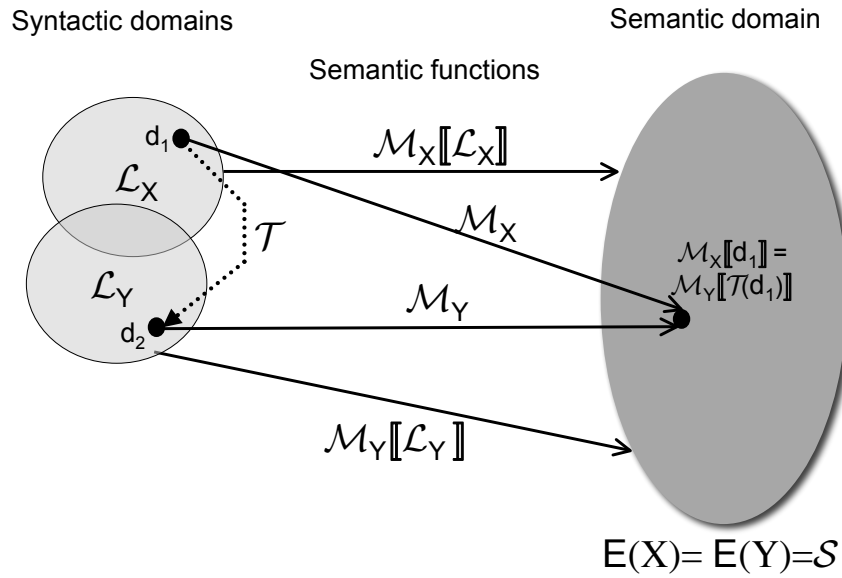
Figure 5.2: Translation between expressively complete languages

One major quality for a language is to be *expressively complete* (Definition 5.2.2). It assures that all the intended meanings can be expressed within the language and that the tools supporting it

will not be restrained at the end. Since languages compete for expressiveness, it often happens that they reach the same maximal expressiveness. It means that they are all *expressively complete* and that there exists a translation between each couple of compared languages (Figure 5.2). Therefore, comparing them according to their expressiveness is not sufficient. This is for instance the case for programming languages, that are almost all Turing-complete and can thus express the same computable functions. Consequently, we need finer criteria than expressiveness to compare these languages.

The idea is to study the *properties of the translations* between those languages:

- Do they preserve the structure of the original diagram?

- Do they increase or reduce the size of the original diagram?

The former property is addressed by the concept of *embeddability* (or naturalness) (Section 5.3) whereas *succinctness* takes care of the second (Section 5.4).

5.3   Embeddability

In theory, when two languages have the same expressiveness, there are two translations back and forth between them that preserve their respective semantics. However, these translations might modify the structure of the original diagram. The structure of a diagram usually corresponds to the shape formed by its nodes and edges. Embeddability (a.k.a naturalness) checks if these translations are *natural*. A translation is said *natural* when it respects the structure of the original diagram (Kleene, 1952; Felleisen, 1990). A *natural* translation is also called an *embedding* (Definition 5.3.1).

Definition 5.3.1 (Embedding) *An* embedding *is a* translation *(Definition 5.2.3)* $T : \mathcal{L}_X \rightarrow \mathcal{L}_Y$ *that is* compositional *(Definition 5.3.2).*

Definition 5.3.2 (Compositional Translation) *A translation of a sentence is* compositional *(or* modular *or* homomorphic *or* natural*) when it is determined by the translations of its constituent parts, and the way in which those parts are combined. The translation* $\mathcal{T} : \mathcal{L}_X \rightarrow \mathcal{L}_Y$, *is compositional when* $\mathcal{T}(C_1(\vec{x})) = C_2(\mathcal{T}(\vec{x}))$, *for all constructs* C_1 *of* $\mathcal{L}_X$, *where* $\vec{x}$ *is a vector of meta-variables, adequately typed with respect to* C_1*'s sub-terms expected types.* C_2 *is an expression in* $\mathcal{L}_Y$ *containing the same meta-variables.* C_2 *is usually noted* $\mathcal{T}(C_1)$.

Example 5.3.1 *A typical example is the compositional translation between programming languages "Pascal" (Mickel et al., 1991) and "Pascal without* **for***". The translation is immediate for all the constituent parts of a sentence (program) written in Pascal except for the* **for** *instruction that should be translated according to the translation provided in Table 5.1 where* i, l, h, S *are meta-variables.*

However, the sentences to translate are not only textual. In the following, we distinguish textual (Section 5.3.1) and visual languages (Section 5.3.2)

Instead of ...	write ...

for i := 1 **to** h **do** S	i := 1;
	 while i <= h **do**
	      begin
	          S;
	          i:=i+1
	      end;

Table 5.1: Translation in Pascal: *for* into *while*

5.3.1   Textual Languages

For *textual languages*, the requirement to preserve string structure (i.e. embeddability (Definition 5.3.3)) has been called macro-eliminability and inspired by (Kleene, 1952; Felleisen, 1990). Macro-eliminability relies on the assumption that the concerned textual languages have a context-free grammar (Definition 5.3.4), which in turn enables to define the translation in a compositional way (i.e as an embedding (Definition 5.3.1)).

Definition 5.3.3 (Embeddability) *A Context-Free language X is* embeddable *into Y iff there is an embedding (Definition 5.3.1) from $\mathcal{L}_X$ to $\mathcal{L}_Y$.*

Definition 5.3.4 (Context-Free Grammar) *A Context-Free Grammar G is a 4-tuple:*
$G = (V_t, V_n, Pr, S)$ *where*

- V_t *is a finite set of terminals;*

- V_n *is a finite set of non-terminals;*

- *Pr is a finite set of production rules where a production rule is of the form $V_n \rightarrow (V_t \cup V_n)^*$;*

- S *is an element of V_n, the distinguished starting non-terminal.*

Example 5.3.2 *This example illustrates Definition 5.3.4 by providing a simple Context-Free Grammar with an alphabet composed of two symbols: "a","b" that contains a different number of "a" and "b". This Context-Free Grammar G is a 4-tuple: $G = (\{a, b\}, \{S, A, B, T\}, Pr, S)$ where[*]*

$$Pr \quad = \{ \quad S \rightarrow A|B, \\ A \rightarrow TaA|TaT, \\ B \rightarrow TbB|TbT, \\ T \rightarrow aTbT|bTaT|\epsilon \, \}$$

More real examples of Context-Free Grammars can be found in the literature. For instance, it exists a definition in BNF of the "Pascal" programming language Context-Free grammar in (Mickel et al., 1991, p215).

---

[*]ϵ stands for the empty string.

5.3.2 Visual Languages

For *visual languages* (Definition 5.3.5), the notion of context-free grammar is not general enough to be directly applicable. In addition, complex visual languages are not always "context-free" as, for instance, relative positions in multi-dimensional spaces are meaningful.

Definition 5.3.5 (Visual Languages) *A* visual language *is any language that lets users communicate by manipulating graphical elements in a multi-dimensional space.*

As recalled in (Moody, 2006b), graphical representation allows *visual variables* to encode information such as shape, size, colour, orientation, texture, horizontal and vertical position and their corresponding values. *Textual languages* are visual languages working in one dimension where only the sequence of characters is important. Examples of more complex visual languages are the American Sign Language (ASL) where the temporal dimension is central or dataflow diagram language (Sutherland, 1966), FD languages (Kang et al., 1990) and UML (OMG, 2008) where labelled graphs are essentially used to represent their syntactic domain. Once the syntactic domain of a language is a graph, the definitions of embeddability and embedding can be adapted for Node-Labelled graphs (Definition 5.3.6) (Janssens, 1983). We will call them graphical embeddability and graphical embedding.

Definition 5.3.6 (Node-Labelled Graph (Janssens, 1983)) *A Node-Labelled Graph G is a 4-tuple: $G = (N_G, E_G, L_G, \lambda_G)$ where*

- *N_G is a finite non-empty set of labelled nodes;*

- *E_G is a finite set of edges where an edge is a couple of elements in N_G;*

- *L_G is a finite non-empty set of labels;*

- *λ_G is a labelling function from N_G into L_G.*

In the following, we revisit the classical definitions of embeddability (Definition 5.3.3) and embedding (Definition 5.3.1). We propose definitions for *graphical embeddability* (Definition 5.3.9) and *graphical embedding* (Definition 5.3.10) that generalise the definitions of embeddability and embedding. We follow and adapt the ideas presented in (Janssens and Rozenberg, 1980).

In (Janssens and Rozenberg, 1980; Janssens, 1983) the authors propose a "context-free" grammar for graph languages called Node-Label Controlled (NLC) graph grammar (Definition 5.3.7). The NLC grammar is analogous to the rewriting performed in classical string grammars. Nevertheless, *productions* in NLC grammar are more complex. Indeed a production is of the form $M \rightarrow D$ where a *mother graph* (M) is replaced by a *daughter graph* (D) in the *host graph* (H). The new host graph is named $\overline{H}$. Therefore, comparatively with string grammars *graph rewriting* is not sufficient. Daughter graphs should be correctly reconnected to the host graph. This reconnection, called *graph embedding* in (Janssens, 1983), is provided through a set of connection relations (C) for the whole grammar rather than for each node separately. The production is said *node-controlled* when the mother graph (M) (left part of the production) is restricted to an element of the total alphabet (A).

Definition 5.3.7 (NLC Grammar (Janssens, 1983)) *A NLC grammar G is a 5-tuple:*
$G = (A, T, Pr, C, Z)$ *where*

- *A is a finite non-empty set, called the total alphabet;*

- *T is a non-empty subset of A, called the terminal alphabet;*

- *Pr is a finite subset of $A \times W$, called the set of productions, where W is the set of all Node-Labelled Graphs over A;*

- *C is a subset of $A \times A$, called the connection relation;*

- *Z is a Node-Labelled Graph over A called the axiom.*

Example 5.3.3 *This example illustrates Definition 5.3.7. It provides a Node-Label Controlled Grammar with an alphabet composed of five labels: a, b, c, d, e. They are all terminal. The production defined translates each node labelled by e into four nodes (two labelled by c, one labelled by b and the last labelled by d) and four edges connecting these new nodes. In addition, two (re)connections have been defined: $\{(b, a), (c, d)\}$. The last (re)connection $((c, d))$ means that if one new node labelled by c is generated by a production, this new node should be connected to every node labelled by d. This is only possible if it exists a node labelled by d in the graph before and after the trigger of the production.*

Let G be the NLC Grammar of the form $G = (A, T, Pr, C, Z)$ where

- $A = \{a, b, c, d, e\};$

- $T = \{a, b, c, d, e\};$

- $Pr = \{(\ e,$ $)\};$

- $C = \{(b, a), (c, d)\};$

- $Z =$
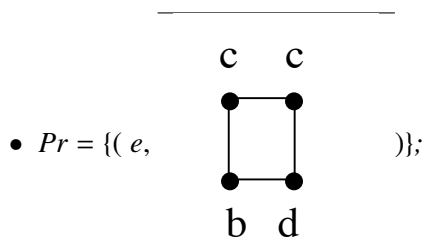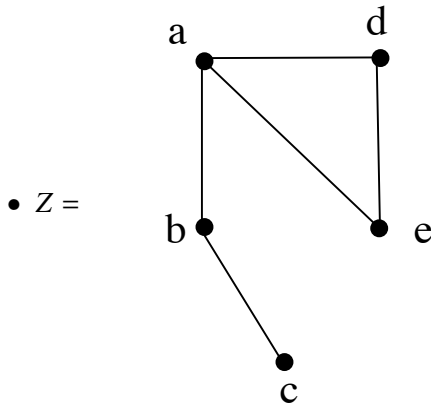
In Table 5.2, the single production rule in Pr is applied to the application node labelled by e that is circled. This node with its connected edges are eliminated in $\overline{H}$ and replaced by four new nodes and four new edges. In addition, three new edges are added to replace the previous edges connected to this node labelled by e. According to the definition of the (re)connections (C), both new nodes labelled by c are connected to the node labelled by d and the new node labelled by b is connected to the node labelled by a.

Table 5.2: Example: Production in NLC Grammar adapted from (Janssens, 1983)

Once we have recalled the work presented in (Janssens and Rozenberg, 1980), we now underline how we adapt it to define graphical embeddings:

- Graphical embeddings are defined over visual languages with syntactic domains defined as Node-labelled Directed Acyclic Graphs (NLDAGs) (Definition 5.3.8).

Definition 5.3.8 (Node-Labelled Directed Acyclic Graph) *A Node-Labelled Directed Acyclic Graph G is a Node-Labelled Graph $(N_G, E_G, L_G, \lambda_G)$ (Definition 5.3.6) where*

- *G is directed: E_G is a subset of $(N_G \times N_G) \setminus \{(x, x)|x \in N_G\}$, with $(x, y) \in E_G$, (x, y) alternatively noted: $x \to y$.*

- *E_G is acyclic: $\nexists n_1, ..., n_k \in N_G . n_1 \to ... \to n_k \to n_1$.*

- Graphical embeddings are not limited to translations within the same language. They may translate graphs from one visual language (X) to another (Y) with different syntactic domains ($\mathcal{L}_X$, $\mathcal{L}_Y$) and eventually different labels ($L_X \neq L_Y$).

- Graphical embeddings should not be confused with "graph embeddings" in (Janssens and Rozenberg, 1980) that only concerns the reconnection of the edges previously related to the defined node.

- Graphical embeddings are of the form $M \to D$ with a mother NLDAG (M) and its daughter NLDAG (D). M and D respectively contain more information than in (Janssens and Rozenberg, 1980):

  - *M contains the defined node (N), all the nodes connected to it and the edges connecting them.*

  - *D contains the new node(s) that will replace the defined node and in addition new edge(s) that will reconnect the new node(s) to the nodes previously connected to the defined node in M.*

Definition 5.3.9 (Graphical embeddability) *A visual language X is* embeddable *into Y iff there is a* graphical embedding *(Definition 5.3.10) from $\mathcal{L}_X$ to $\mathcal{L}_Y$.*

Definition 5.3.10 (Graphical embedding) *A* graphical embedding *is a linear* translation *(Definition 5.2.3)* $\mathcal{T} : \mathcal{L}_X \to \mathcal{L}_Y$ *that is* node-controlled *(Janssens and Rozenberg, 1980) (Definition 5.3.11).*

Definition 5.3.11 (Node-Controlled Translation) *The translation of a diagram is* node-controlled *(or compositional on graphs) when it is determined by translations of its constituent nodes, and the way in which edges are reconnected within the diagram (Janssens and Rozenberg, 1980). A* node-controlled *translation $\mathcal{T}$ is of the form $\mathcal{T} : M \to D$, where:*

- *M is a NLDAG called* mother diagram *of the form $M = (\{N\} \cup N_M, E_M, \{L_N\} \cup L_M, \lambda_M)$ where:*

  - *$N \notin N_M$ is the node called the* defined node*;*

  - *N_M is the set of nodes connected to N;*

  - *$E_M \subseteq ((\{N\} \times N_M) \cup (N_M \times \{N\}))$ is the set of all possible incoming and outgoing edges related to the node N;*

  - *L_N is the label of N;*

  - *L_M is the set of labels associated to the nodes connected to N;*

  - *λ_M is a labelling function from $\{N\}$ into $\{L_N\}$ and from N_M into L_M.*

- *D is a NLDAG called the* daughter diagram *of the form $D = (N_D \cup N_M, E_D, L_D \cup L_M, \lambda_D)$ where:*

  - *N_D is the set of new nodes with $N_D \cap (N_M \cup \{N\}) = \emptyset$;*

  - *$E_D \subseteq (N_M \cup N_D) \times (N_D \cup N_M)$ is the set of new edges;*

  - *L_D is the set of labels associated to new nodes in N_D;*

  - *λ_D is a labelling function from N_D into L_D and from N_M into L_M.*

The node-controlled translation ($\mathcal{T} : M \rightarrow D$) replaces the application node (S) by a NLDAG composed of new nodes (N_D) and new edges (E_D). An application node is a node in the host graph ($S \in N_H$) that has the same label than the *defined node* (N) in M: $\lambda_H(S) = \lambda_M(N)$. Hence, the translation of the host NLDAG (H) results in a new NLDAG ($\overline{H}$) where $N_{\overline{H}} = (N_H \setminus \{S\}) \cup N_D$, $E_{\overline{H}} = (E_H \setminus E_M) \cup E_D$ and $L_{\overline{H}} = (L_H \setminus \{L_M\}) \cup L_D$.

Example 5.3.4 *Consider the translation $\mathcal{T}$ between two different languages with respectively syntactic domain $\mathcal{L}_X$ and $\mathcal{L}_Y$. The only difference between these languages is that they do not have the same sets of labels: $L_2 \in L_X$ while $L_2 \notin L_Y$. Hence, to translate a diagram $\mathcal{L}_X$ to $\mathcal{L}_Y$ we need to translate nodes labelled by L_2. In Figure 5.3 we illustrate such a translation (named $r : M \rightarrow D$) where each node labelled by L_2 is simply eliminated while its incoming and outgoing edges are reconnected. Its mother NLDAG (M) is composed of a defined node (N) labelled by L_2 ($\lambda_X(N) = L_2$) and connected to nodes ($X_1, \ldots, X_{j+i}$) by incoming ($e_1, \ldots, e_j$) and outgoing ($e_{j+1}, \ldots, e_{j+i}$) edges (E_M). Its daughter NLDAG (D) contains no new nodes ($N_D = \emptyset$) but many new edges (E_D) reconnecting remaining nodes ($X_1, \ldots, X_{j+i}$). For this reconnection, each origin of the incoming edges of N are reconnected to each destination of the outgoing edges of N. Formally, the translation r is of the form $M \rightarrow D$ where:*

- *$M = (\{N\} \cup \{X_1, \ldots, X_{j+i}\}, \{(X_1, N), \ldots, (X_j, N)\} \cup \{(N, X_{j+1}), \ldots, (N, X_{j+i})\}, \{L_2\} \cup L_M, \lambda_M)$;*

- *$D = (\{X_1, \ldots, X_{j+i}\}, \{(X_1, X_{j+1}), \ldots, (X_1, X_{j+i})\} \cup \ldots \cup \{(X_j, X_{j+1}), \ldots, (X_j, X_{j+i})\}, L_M, \lambda_D)$*

Table 5.3 illustrates an application of the translation r on the application node n_2 *since $\lambda(n_2) = L_2$. The host NLDAG (H) is composed of four nodes n_1, n_2, n_3, n_4 and three edges (n_1, n_2), (n_2, n_3), (n_2, n_4). The resulting NLDAG ($\overline{H}$) is composed of three nodes n_1, n_3, n_4 and two edges (n_1, n_3), (n_1, n_4). Formally:*

- *$H = (\{n_1, n_2, n_3, n_4\}, \{(n_1, n_2), (n_2, n_3), (n_2, n_4)\}, \{L_1, L_2, L_3\}, \lambda_H)$*

- *$\overline{H} = (\{n_1, n_3, n_4\}, \{(n_1, n_3), (n_1, n_4)\}, \{L_1, L_3\}, \lambda_{\overline{H}})$*

The graphical definitions given for embeddability and embedding generalise the original ones. As illustrated in the following Example 5.3.5, every syntax tree can be viewed as a labelled tree. Moreover, if syntax trees in context free grammars allow sharing (that means that they are graphs), then the graphical embedding is always linear in graph size.

$$r: \quad M \quad \longrightarrow \quad D$$

$X_1 \quad X_j$

$e_1 = (X_1, N) \qquad e_j = (X_j, N)$

$\lambda(N) = L_2 \quad N$

$e_{j+1} = (N, X_{j+1}) \qquad e_{j+i} = (N, X_{j+i})$

$X_{j+1} \quad X_{j+l}$

$X_1 \quad X_j$

$e_{1,j+1} = (X_1, X_{j+1}) \qquad e_{j,j+1} = (X_j, X_{j+1})$

$X_{j+1} \quad X_{j+l}$

$N_M = \{X_1, ..., X_{j+l}\}$
$E_M = \{e_1, ..., e_{i+j}\}$
$L_N = \{L_2\}$

$N_D = \emptyset$
$E_D = \{(X_m, X_n)\}$ $1 \le m \le j < n \le j+i$
$L_D = \emptyset$
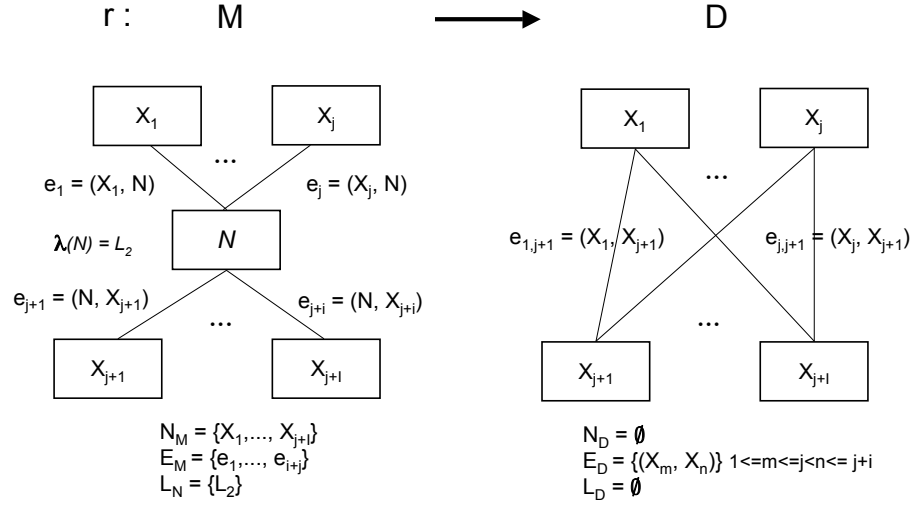
Figure 5.3: An example of a Node-controlled Translation

$H \in NLDAG_X$	$\overline{\overline{H}} \in NLDAG_Y$
$\lambda(n_1) = L_1 \quad n_1$	$\lambda(n_1) = L_1 \quad n_1$
$\lambda(n_2) = L_2 \quad n_2$	
$n_3 \qquad n_4$	$n_3 \qquad n_4$
$\lambda(n_3) = L_3 \qquad \lambda(n_4) = L_3$	$\lambda(n_3) = L_3 \qquad \lambda(n_4) = L_3$

Table 5.3: An example of application of the translation r to the node n_2

Example 5.3.5 *A program written in Pascal can also be represented as a syntax DAG (NLDAG). Hence, the textual translation between* for *and while instructions, presented in Table 5.1, can be understood as a graphical embedding (Table 5.4). The node (N) labelled by* for *is the* defined *node and is connected to nodes* i, l, h, S. *In the tree form,* i *is duplicated by the translation, and if* i *is big, this can blow up the translation. In the graph form,* i *cannot be duplicated since it is allowed to occur only once in D. But we can connect the nodes of D (* n_2, n_4, n_7 *and* n_8 *) several times to* i. *Thus we obtain the same net effect without blow-up. Hence, "Pascal" is embeddable into "Pascal without* **for***". Since the defined node N and its outgoing edges are replaced by a single rooted NLDAG, its incoming edges should be reconnected to the node (* n_1 *) that has no parents at the top of D.*
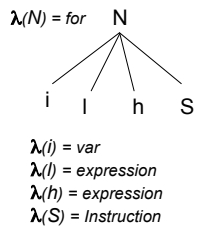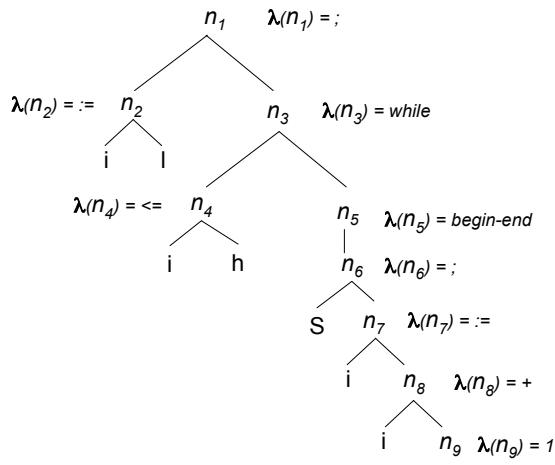
Instead of ...	write ...	Expansion Factor
$\lambda(N) = for$ N i l h S $\lambda(i) = var$ $\lambda(l) = expression$ $\lambda(h) = expression$ $\lambda(S) = Instruction$		5

Table 5.4: Graphical embedding in Pascal: *for* into *while*

The graphical convention illustrated in Table 5.4 will be used throughout the thesis to define graphical embeddings. The defined node and its associated edges (*M*) are represented in the first column (*Instead of ...*) of the table while the replacing nodes and their associated edges (*D*) are represented in the second column (*write...*). A third column (*Expansion Factor*) is added to indicate what are the impacts of this embedding on the size of the NLDAG. This expansion factor will be useful during succinctness analysis (see Section 5.4). To avoid overloading of the table, we consider that if the incoming edges of the defined node (and how they are reconnected in *D*) are not explicitly represented then these incoming edges are implicitly reconnected to each node that has no parents in *D*.

Embeddings are of practical relevance because they ensure that it exists a translation from one language to another that preserves the whole shape of the diagrams and generates only a linear increase in size. Traceability between two diagrams is then greatly facilitated and tool interoperability is made more transparent. Furthermore, controlling the size of diagrams helps avoiding tractability issues for reasoning algorithms taking the diagrams as input (Section 5.1).

Embeddability can also exist between a language and a subset of itself. A language that is non-trivially self-embeddable is called *harmfully redundant* (Definition 5.3.12). A language is self-embeddable when there is a graphical embedding inside the same language. This means that it is unnecessarily complex. All diagrams can be expressed in the simpler sub-language without loss of structure and with only a linear increase in size.

Definition 5.3.12 (Harmful redundancy) *A language X is* harmfully redundant *iff there is a construct c in X that has a graphical embedding in X \ c.*

Example 5.3.6 *The Pascal programming language is harmfully redundant because the* **for** *instruction can be linearly translated into the same language without* **for** *(see Table 5.4). Hence, it exists a self-embedding.*

Nevertheless, every decently natural language is *harmlessly redundant* (Definition 5.3.13). Indeed, when two different diagrams share the same meaning they are semantically equivalent but visually different. This situation is illustrated in Figure 4.1 where two different diagrams (*herDiagram* and *hisDiagram*) in the syntactic domain of the language map to the same element in the semantic domain ($\mathcal{M}(herDiagram) = \mathcal{M}(hisDiagram)$).

Definition 5.3.13 (Harmless redundancy) *A language X is* harmlessly redundant *iff $\exists d_1, d_2 \in \mathcal{L}_X.d_1 \neq d_2 \wedge \mathcal{M}_X[\![d_1]\!] = \mathcal{M}_X[\![d_2]\!] \wedge \nexists c \in \mathcal{L}_X.c$ has a graphical embedding in X \ c.*

Example 5.3.7 *If no self-embeddings exists in the programming language "Pascal without* **for***" then this language is* harmlessly redundant*. Indeed, two different programs written in "Pascal without* **for***" may share the same meaning (see Table 5.5). Here, two programs share the same meaning when they respectively end with the same memory state.*

hisProgram	herProgram

x:=0;
i:=4;
x:=x+1:
x:=x+2:
	x := 0;
	i := 1;
	while i <= 3 **do**
	       begin
	           x:=x+1;
	           i:=i+1;
	       end;

Table 5.5: Two semantically equivalent programs in Pascal

5.4 Succinctness

However, linear translations are not always possible. In this case, the blow-up in size of the diagram must be measured. The size of a diagram is usually measured by the number of its nodes and edges. This is achieved by examining succinctness (Definition 5.4.1) that is a coarser-grained criteria.

For languages with the same expressiveness, embeddability guarantees that their respective diagrams are (roughly) of the same size (since there exists a linear translation). Sometimes, languages of the same expressiveness cannot be compared by embeddings since they are a very refined comparison. *Succinctness* enables to compare the size of their respective diagrams by computing the size of the diagrams before and after translation from one language to the other. *Succinctness* measures the blow-up caused by a change of language.

Definition 5.4.1 (Succinctness) *Let $\mathcal{G}$ be a set of functions from $\mathbb{N} \rightarrow \mathbb{N}$. A language X is $\mathcal{G}$-as succinct as Y, noted $Y \leq \mathcal{G}(X)$, iff there is a translation $\mathcal{T} : \mathcal{L}_X \rightarrow \mathcal{L}_Y$ that is within $\mathcal{G}$: $\exists g \in \mathcal{G}, \forall n \in \mathbb{N}, \forall d \in \mathcal{L}_X, |d| \leq n \Rightarrow |\mathcal{T}(d)| \leq g(n)$. Common values for $\mathcal{G}$ are "identically" = $\{n\}$, "thrice" = $\{3n\}$, "linearly" = $O(n)$, "cubically" = $O(n^3)$, "exponentially" = $O(2^n)$. We will omit "identically".*

Example 5.4.1 *If sharing is allowed in the second column of Table 5.4 then the expansion factor of this embedding is 5. Indeed, the defined Node (N) in M and its four outgoing edges are replaced by nine nodes ($n_1, \ldots, n_9$) and fifteen edges. If a program written in "Pascal" contains n **for** instructions then its translation into a program written in "Pascal without **for**" increases at maximum the size of the program by n.5. Hence, "Pascal" is linearly-as succinct as "Pascal without **for**" ("Pascal without **for**" $\leq O($"Pascal"$)$) and conversely "Pascal without **for**" is identically-as succinct as "Pascal" since "Pascal without **for**" is a sub-language of "Pascal".*

If the language X is $\mathcal{G}$-*as succinct as* Y, this might entails that $\mathcal{L}_X$'s diagrams are likely to be more "readable". Also, if one needs to translate diagrams from $\mathcal{L}_X$ to $\mathcal{L}_Y$[†], succinctness will be an indicator of the difficulty to maintain traceability between the original and the generated diagram. We should note that traceability is hard to measure precisely because succinctness does not provide information on the structure of the generated diagrams. However, looking at the translation will provide the information. In this sense, succinctness is a coarser-grained criteria than embeddability. Finally, as we already pointed out for embeddability, increases in size are generally not good for the tractability of algorithms (Section 5.1).

5.5 Chapter Summary

Throughout this chapter, we have defined and discussed formal criteria to compare languages in a rigorous yet natural way. These criteria are *complexity*, *expressiveness*, *embeddability* and *succinctness*. Their main advantages are to be (1) language-independent, (2) user-independent, (3) formal and (4) based on strong theories such as language theory (Hopcroft et al., 2000) and computational complexity theory (van Leeuwen, 1990; Papadimitriou, 1994). Complexity is usually linked to

---

[†]E.g., because a tool for achieving some desired functionality is only available in language *Y*

expressiveness and succinctness. In general, the more a language is expressive, the more its decision problems are complex. In addition, more succinct problems tend to have higher complexity. A compromise between the language expressiveness or succinctness and the complexity of its decision problems should be devised.

This chapter concludes Part II where we have described our approach, its purpose, central concepts and scope according to SEQUAL framework. In Part III, we will apply and refine it to evaluate FD languages. We will survey existing FD languages (Chapter 6) and study them according to an explicit method. This method (Chapter 7) will allow us to (1) formally define FD languages (Chapter 8) and to (2) compare and evaluate them (Chapter 9).

Part III

Quality of Feature Diagram Languages

Chapter 6

FD Languages: State of the Art

In Part II, we have described SEQUAL as a quality framework to evaluate model and language quality. We have refined it according to principles and criteria related to formal languages. Throughout this chapter, we present a state of the art of the existing FD languages, underlining their main particularities. Afterwards, we will describe, in Chapters 7 and 8, a method to evaluate them according to the formal criteria defined in Chapter 5. Finally, we will apply this method and provide the main results concerning FD languages in Chapter 9.

In the last 15 years or so, research and industry have developed several FD languages. The first and seminal proposal was introduced as part of the FODA method back in 1990 (Kang et al., 1990). An example of a FODA FD inspired from a case study defined in (Cohen et al., 2002a) is given in Figure 6.1.
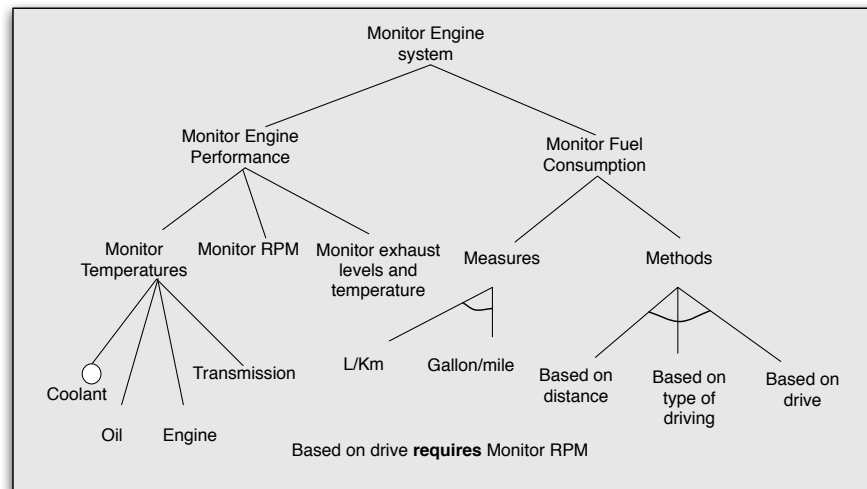
Figure 6.1: FODA (OFT): Monitor Engine System

This FODA FD indicates the allowed combinations of features for a family of systems intended to monitor car engines. FODA features are nodes of a graph represented by strings and related by various types of edges. On top of the figure, the feature `Monitor Engine System` is called the *root feature*, or *concept*. The edges are used to progressively decompose it into more detailed features. The various kinds of edges and their exact meaning will be discussed extensively throughout the next chapters.

In the sequel, we will refer to FODA FD as Original Feature Trees or OFT for short. "Original" is because OFTs were the first FD language ever proposed and because they were the starting point for the work of many researchers. "Tree" is because in OFT, FD are structured as trees *vs.* single-rooted directed acyclic graphs (DAG) (Kang et al., 1998). Accordingly and for convenience, we have developed a naming scheme to designate the reviewed FD languages (Table 6). FD languages are named with three or four letters acronyms. The first or first two letters are representative of the original name of the language, the method it comes from or one of its authors. Then comes an 'F' (for feature). The last letter is either 'D' (for DAG) or 'T' (for trees).

Since Kang *et al.*'s initial proposal, several extensions have been devised as part of the following methods: FORM (Kang et al., 1998), FeatureRSEB (Griss et al., 1998), Generative Programming (Eisenecker and Czarnecki, 2000), FORE (Riebisch et al., 2002; Riebisch, 2003) , PLUSS (Eriksson et al., 2005), and in the work of the following authors:(van Gurp et al., 2001; van Deursen and Klint, 2002; Mannion, 2002; Cechticky et al., 2004; Czarnecki et al., 2005c,b; Batory, 2005; Benavides et al., 2005a; Wang et al., 2005a; Sun et al., 2005; Wang et al., 2005b; Asikainen et al., 2006; Janota and Kiniry, 2007). First proposals of FD languages (Kang et al., 1998; Griss et al., 1998; Eisenecker and Czarnecki, 2000; Riebisch et al., 2002; Riebisch, 2003; van Gurp et al., 2001; Eriksson et al., 2005) were not formally defined. Recently some authors have proposed formal definitions for FDs (van Deursen and Klint, 2002; Mannion, 2002; Bontemps et al., 2004; Cechticky et al., 2004; Czarnecki et al., 2004, 2005b; Batory, 2005; Benavides et al., 2005a; Wang et al., 2005a; Sun et al., 2005; Wang et al., 2005b; Schobbens et al., 2007; Asikainen et al., 2006; Janota and Kiniry, 2007).

The structure of this chapter is as follows. First, we describe in Section 6.1 the method followed to gather the FD languages to study and to define the scope of our survey. Then, we introduce in Section 6.2 an example on which we illustrate each selected FD language. Finally, we distinguish and present informal and formal FD languages, respectively in Sections 6.3 and 6.4.

6.1   Survey Method

The purpose of this survey was to review and analyse the literature in order to elicit the relevant proposals concerning FD languages. Primary proposals of FD languages were identified from Citeseer, Google Scholar, ACM and IEEE databases. Bibliographical references and cross-references were used to discover other proposals. A proposal was selected when either it presents and discusses the basic constructs of the proposed FD language or it presents and justifies the various extensions it provides with respect to another FD language.

During this survey, we have mainly looked for (1) differences between FD languages, (2) descriptions of their syntax and semantics and (3) methods and tools supporting them. Among these

Acronyms	Languages	Language references
BFT	Batory Feature Tree language	Batory (2005)
EFD	Extended Feature Diagram language	Riebisch et al. (2002)
FFD	Free Feature Diagram language	Schobbens et al. (2007)
FoFD	Forfamel Feature Diagram language	Asikainen et al. (2006)
GPFT	Generative Programming Feature Tree language	Eisenecker and Czarnecki (2000)
OFD	FORM Feature Diagram language	Kang et al. (1998)
OFT	FODA Feature Tree language	Kang et al. (1990)
PFT	PLUSS Feature Tree language	Eriksson et al. (2005)
RFD	RSEB Feature Diagram language	Griss et al. (1998)
VBFD	van Gurp and Bosch Feature Diagram language	van Gurp et al. (2001)
vDFD	van Deursen and Klint Feature Diagram language	van Deursen and Klint (2002)
VFD	Varied Feature Diagram language	Bontemps et al. (2004)
XFD	XML-based Feature Diagram language	Cechticky et al. (2004)

Table 6.1: FD Languages: Acronyms

proposals, we have distinguished two types of FD languages: the formal and the informal FD languages. A language is considered as formal if it possesses mathematical definitions for its syntax and semantics (Harel and Rumpe, 2000, 2004). The implementation of these definitions should drive to minor adaptations. Conversely, a language is considered as informal when no such mathematical definitions are provided. Hence, a language supported by a tool is not necessarily formal. It guarantees that the language has a semantics but does not assure that this semantics has been, previously, formally defined. In addition, there is no absolute evidence that an informal or even a formal semantics is correct with respect to intuition. The first purpose of models is to be understandable by all stakeholders, therefore informal sketches of the (formal) models are also needed.

6.2   Survey Illustration

To illustrate the different FD languages introduced in the next sections, we reuse and simplify an example proposed at the SPLC2 Workshop on Techniques for Exploiting Commonality Through Variability Management (Cohen et al., 2002b). Our aim is to use the different FD languages to model the variability of an engine monitor system. This system manipulates and provides crucial information about fuel consumption and engine performances. Various types of engine monitor systems exist, however each of them should be able to monitor the number of Rounds Per Minute (RPM), the oil, engine and transmission temperatures and the exhaust levels and temperatures. Some of them may also monitor the coolant temperature while others do not. In addition, different measures are offered to monitor the fuel consumption and only one must be selected: *L/Km* or *Gallon/mile*. Finally, various methods are offered to monitor the fuel consumption and at least one must be selected. The system may monitor the fuel consumption according to the distance, the type of driving or the effective drive.

In terms of features, we identify for this example one root, eleven primitive features and five

non-primitive features. The root is named `Monitor Engine system`. Among the eleven primitive features:

- one is *optional* (`Coolant`) and five are *mandatory* (`Oil`, `Engine`, `Transmission`, `Monitor RPM`, `Monitor exhaust levels and temperature`);

- three are *or-features* (`Based on distance`, `Based on type of driving` and `Based on drive`) and two are *xor-features* (`L/Km` and `Gallon/mile`).

The five remaining non-primitive features are used to structure feature decomposition in the FD:

- `Monitor Engine Performance` gathers the features related to the monitoring of the engine;

- `Monitor Fuel Consumption` gathers the features related to the monitoring of the fuel consumption;

- `Monitor Temperatures` gathers the features for which the temperature should be checked;

- `Measures` gathers the features corresponding to the allowed type of measures to monitor the fuel consumption;

- `Methods` gathers the features corresponding to the methods used to monitor the fuel consumption.

6.3 Informally Defined FD Languages

In this section, we survey OFT (Kang et al., 1990) and its informal extensions. Originally, these languages are mainly described informally, their semantics being usually introduced by way of examples. We will first describe each of these languages according to its syntax, stressing the main particularities of its concrete and abstract syntax. As we will see, many differences appear in their concrete syntax, e.g. an *xor*-decomposition is sometimes represented by a diamond, sometimes by a triangle and sometimes by another symbol. Although concrete syntax is an important matter in its own (Moody, 2006b), we will not investigate those kinds of issues. Our main interest is to determine which abstract constructs allow distinguishing one FD language from another.

6.3.1 FODA (OFT)

OFT, the first ever FD language, was introduced as part of the Feature Oriented Domain Analysis (FODA) method (Kang et al., 1990). Its main purpose was to capture commonalities and variabilities at the requirement level. As depicted in Figure 6.1, OFT has the following characteristics:

1. A *concept*, a.k.a *root node*, that refers to the complete system (e.g. `Monitor Engine System`).

2. *Features* that are subject to decomposition (see below) and that can be *mandatory* (by default, e.g. `Engine`) or *optional* (with a hollow circle above, e.g., `Coolant`) or *alternative* (under an *xor*-decomposition, e.g. `L/Km`)). The concept is always mandatory.

3. *Relations* between nodes materialised by:

    (a) *decomposition edges* (or *consists-of*): FODA further distinguishes:

        i. *and-decomposition*, e.g., between `Monitor Fuel Consumption` and its sons (`Measures` and `Methods`) indicating that they should both be present in all feature combinations where `Monitor Fuel Consumption` is present.

        ii. and *xor-decomposition*, e.g., between `Measures` and its sons, `L/Km` and `Gallon/mile`, indicating that only one of them should be present in combinations where `Measures` is.

    Decomposition edges form a tree. See (Kang et al., 1990, p64) where the authors underline that: "a feature diagram [...] is an and/or tree of different features".

    (b) *textual constraints*:

        i. *requires*, e.g., one could add in Figure 6.1 the constraint: `Based on drive` *requires* `Monitor RPM`. It indicates that we need to monitor the engine's RPM in order to monitor fuel consumption using a method based on drive. Thus, the former feature cannot be present if the latter is not.

        ii. *mutex*, an abbreviation for "mutually exclusive with", indicates that two features cannot be present simultaneously.

Semantically, the diagram provided in Figure 6.1 differs from the original example. Indeed, the choice when selecting the method to monitor fuel consumption should not be exclusive. One, two or the three different methods can be integrated in the system even if only one is executed at run-time. The confusion appears because FODA does not allow *or*-decomposition.

6.3.2 FORM (OFD)

Kang *et al.* have proposed the Feature-Oriented Reuse Method (FORM) (Kang et al., 1998) as an extension of FODA. Their main motivation was to enlarge the scope of feature modelling. They argue that feature modelling is not only relevant for requirements engineering but also for software design. Hence, they propose to extend OFT. According to our naming scheme, we call this first extension of OFT: OFD for "Original Feature Diagram language".

OFD is frequently used in the literature. There are two variants of it. The simplest just extends OFT with the possibility to draw DAGs (directed acyclic graphs) instead of being limited to trees. The second is more complex as it further adds three new constructs to OFT: layers, implementation and generalisation/specialisation relationships. We only look at the simpler form illustrated in Figure 6.2 where one marginal difference between OFD and OFT appears: feature names in OFD are depicted within boxes while not in OFT. In this work, we consider the constructs that have an influence on the feature combinations allowed in the final products. Therefore, the only relevant (and, as we will see, crucial) change concerns the structure of the FD. Indeed, OFDs are not limited to trees but allow feature sharing, i.e. DAGs. See (Kang et al., 1998, p1) where the authors underline that "a feature model [...] captures commonality as an AND/OR graph".

In our running example, the OFD still forms a tree. Therefore, the FORM version of it (Figure 6.2) is similar to Figure 6.1, except that features are graphically represented within boxes. Still
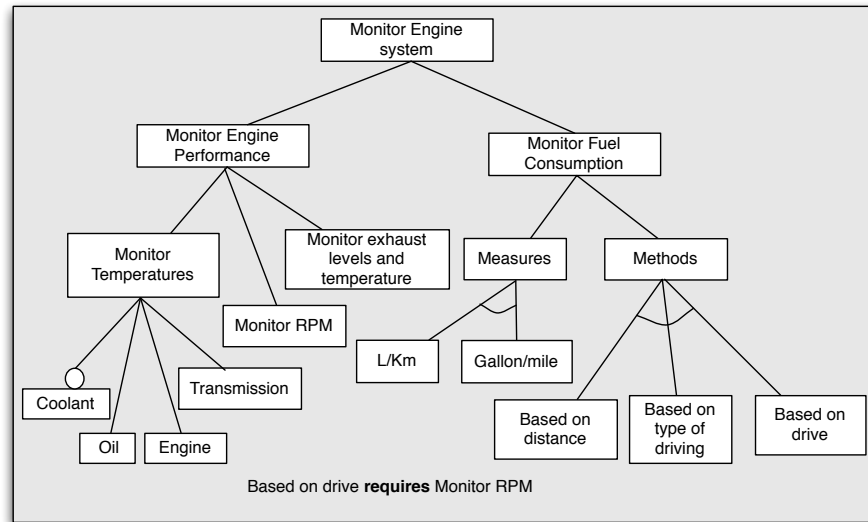
Figure 6.2: FORM (OFD): Monitor Engine System

the diagram in Figure 6.2 does not exactly reflect the intended product line. This time we can bring a solution for OFD. Indeed, since OFD allows feature sharing, an *or*-decomposition could be translated into *and* and *xor*-decompositions. This translation consists in replacing the desired *or*-decomposition under the feature `Methods` by its corresponding *xor*-decompositions for which new intermediate features must be added. This translation is illustrated in Figure 6.3 where the intermediate nodes *Or*1, *Or*2, *Or*3 and *Or*4 are necessary to guarantee the semantic equivalence with the original *or*-decomposition.

6.3.3   FeatuRSEB (RFD)

The FeatuRSEB method (Griss et al., 1998) is a combination of FODA and the Reuse-Driven Software Engineering Business (RSEB) method. RSEB is a use-case driven systematic reuse process, where variability is captured by structuring use cases and object models with *variation points* and *variants*. According to our naming scheme, we call this FD language RFD for "RSEB Feature Diagram language". RFD has the following new characteristics in comparison to OFT:

1. RFDs are DAGs. See (Griss et al., 1998, p6) where a feature model is described as "a feature tree or graph, showing feature names, major relationships and a few attributes."

2. *Or*-decomposition (black diamond) is added to *xor* (white diamond) and *and*-decompositions. Features decomposed with *or* or *xor* are called *variation points* and their sons are called *variants*.

3. Graphical representations are given for the constraints *requires* (leftwards or rightwards dashed arrow) and *mutex* (left right dashed arrow).
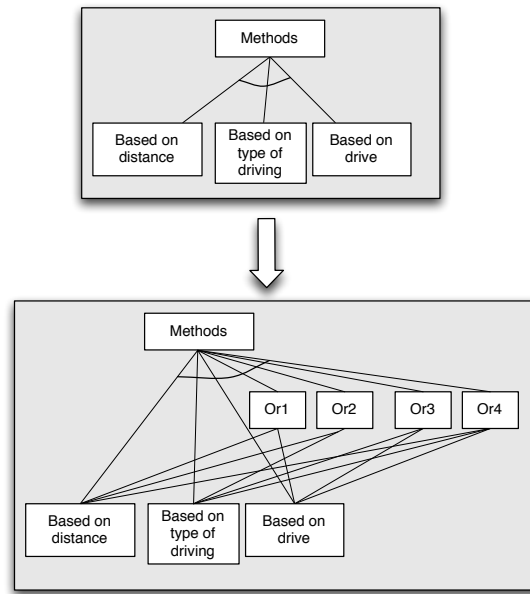
Figure 6.3: FORM (OFD): Or-decomposition translation

Figure 6.4 illustrates how our running example can be modelled using RFD.
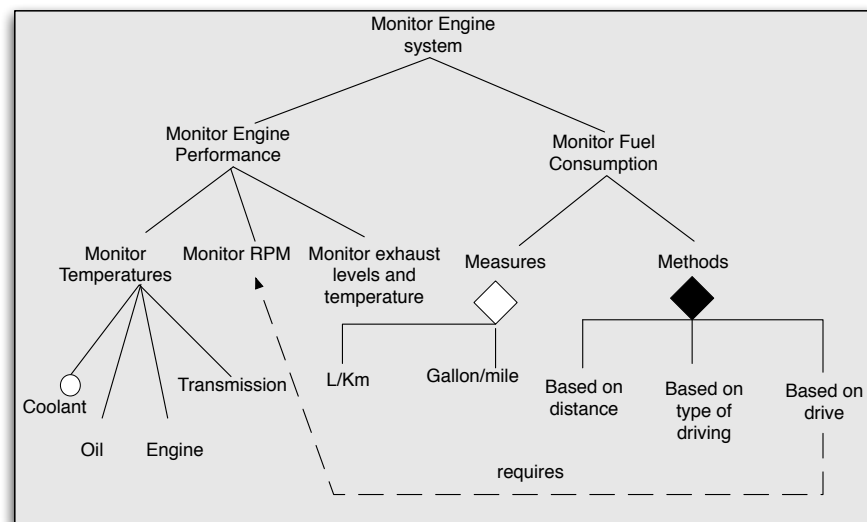
Figure 6.4: FeatuRSEB (RFD): Monitor Engine System

6.3.4   Van Gurp, Bosch and Svahnberg (VBFD)

Van Gurp, Bosch and Svahnberg define another FD language in (van Gurp et al., 2001). According to our naming scheme, we call this FD language VBFD for "van Gurp and Bosch Feature Diagram language".

   This language extends FeatuRSEB (RFD) with *binding times* and *external features*. Binding times indicate when features can be selected and *external features* indicate which technical possibilities are offered by the target platform of the system. VBFD has the following new characteristics in comparison to OFT:

1. VBFDs are DAGs. See (van Gurp et al., 2001, p3) where the authors underline that: "Our extended feature graph is based on the work presented in (Griss et al., 1998)".

2. *Binding times* are used to annotate relationships between features.

3. Features are boxed, as in FORM (OFD).

4. *External features* are represented in dashed boxes.

5. *Or*-decomposition (black triangle) is added to *xor* (white triangle) and *and*-decompositions.

6. Graphical representations (dashed arrows) are given for the constraints *requires* and *mutex*, as in FeatuRSEB (RFD).

 Figure 6.5 illustrates how our running example can be modelled using VBFD.
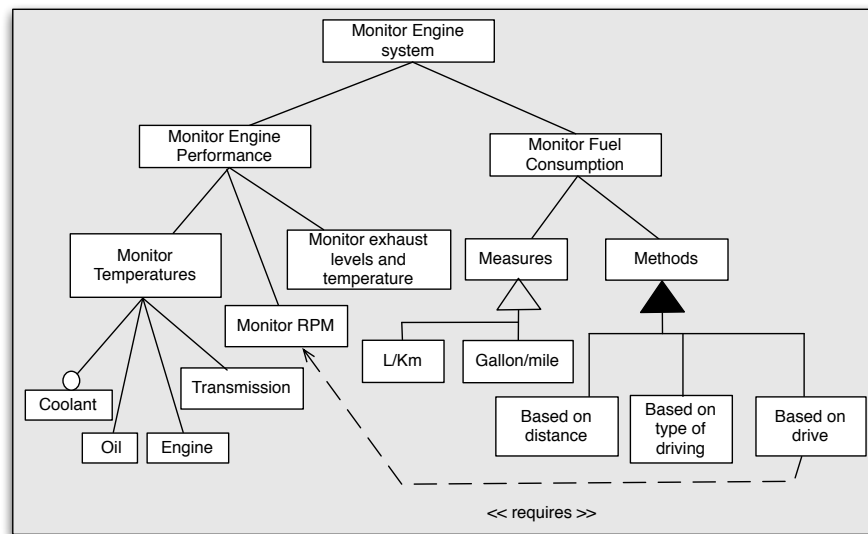
Figure 6.5: VBFD: Monitor Engine System

   In comparison to RFD, these changes essentially concern concrete syntax. Except for the new notions of "binding times" and "external features" that we do not consider, here, since they do not

influence the final combination of features. Hence, we consider the abstract syntax of this language equivalent to the one of RFD.

6.3.5 Generative Programming (GPFT)

Czarnecki and Eisenecker (Eisenecker and Czarnecki, 2000) have studied and adapted FDs in the context of Generative Programming (GP), a new programming paradigm that aims to automate the software development process for product families. For this purpose, they propose a new Feature Tree language extending OFT. See (Eisenecker and Czarnecki, 2000, p87) where the authors underline that: "The nodes and edges [in a feature diagram] form a tree". According to our naming scheme, we call this FD language GPFT for "Generative Programming Feature Trees". GPFT has the following new characteristics in comparison to OFT:

1. *Or*-decomposition (filled cross-cutting curve) is added to *xor* (cross-cutting curve) and *and*-decompositions.

2. Mandatory features are decorated with a filled circle in contrast with the hollow circle used for optional ones.

Figure 6.6 illustrates how our running example can be modelled using GPFT.
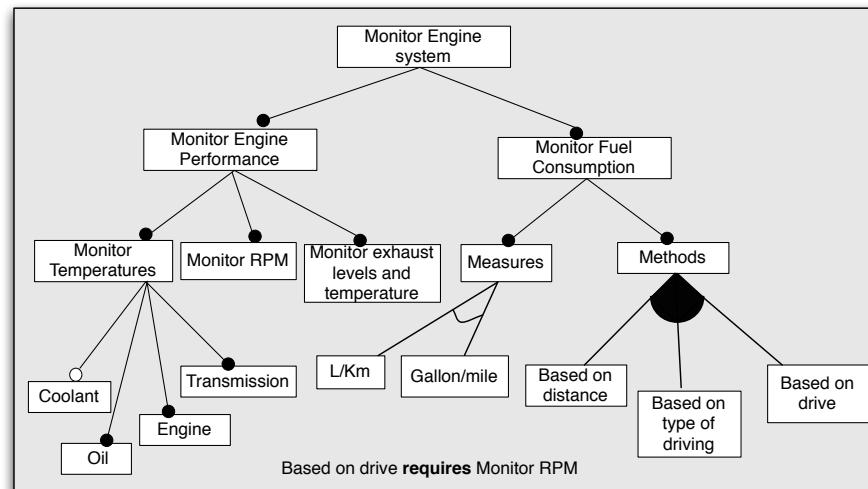
Figure 6.6: Generative Programming (GPFT): Monitor Engine System

Recently, the authors have further augmented their approach with concepts such as *staged configuration* (Czarnecki et al., 2005c), distinguishing between *group* and *feature cardinalities* and formalising their language in (Czarnecki et al., 2005b). This work is shortly described in Section 6.4.5.

6.3.6   FORE (EFD)

Riebisch *et al.* claim that multiplicities (a.k.a. cardinalities) are only partially represented with the previous notations (Riebisch et al., 2002). Moreover, they argue that *"combinations of mandatory and optional features with alternatives, or and xor relations could lead to ambiguities"* (Riebisch, 2003, p.4). As we will see, those "ambiguities" are due to a different conception of what "mandatory" and "optional" mean, and better termed redundancy (see Section 9.1.3.1).

In order to limit these drawbacks, the authors replace *or* and *xor* operators by UML-like *multiplicities* (Riebisch et al., 2002) (a.k.a group cardinalities (Czarnecki et al., 2005b)). Multiplicities consist of two integers: a lower and an upper bound. Multiplicities are illustrated in Figure 6.7 for the decomposition of features `Measures` and `Methods`. In fact, they are used for all decompositions but not mentioned explicitly. For instance, when both lower and upper bounds equal the number of sons, this decomposition with multiplicities is equivalent to an *and*-decomposition.

The second particularity of this language concerns the edges that can be mandatory or optional. The first ones are marked with a filled circle at the lower end and the second ones are marked with a hollow circle at the lower end. This should be underlined as most other FD languages prefer to mark optionality on nodes rather than on edges. As illustrated in Figure 6.8, optional edges are convenient when DAGs are used since a shared feature can be optional on one side and mandatory on another.

According to our naming scheme, we call this FD language EFD for "Extended Feature Diagram language". EFD has the following new characteristics in comparison to OFT:

1. EFDs are DAGs. See (Riebisch et al., 2002, p3) where the authors underline that: "A feature is a node in a directed-acyclic graph. Relationships between features are expressed by edges [...]."

2. The *xor*, *or* and *and*-decompositions are replaced by the more general *card*-decomposition.

3. Optional and mandatory features are replaced by optional (lines terminated by a hollow circle) and mandatory edges (lines terminated by a filled circle).

4. Graphical representations (dashed arrows) are given for the constraints *requires* and *mutex*.

6.3.7   PLUSS (PFT)

The Product Line Use case modelling for System and Software engineering (PLUSS) approach (Eriksson et al., 2005) is based on FeatuRSEB. It combines FDs and use case diagrams to depict the high level view of a product family. The authors propose a new Feature Tree language based on Mannion's proposal (Mannion, 2002). See (Eriksson et al., 2005, p2) where the authors underline that: "In feature models, features are organized into trees of AND and OR nodes that represent the commonalties and variations in the modeled domain.". According to our naming scheme, we call this FD language PFT for "PLUSS Feature Tree language". As illustrated in Figure 6.9, the main characteristics of PFT in comparison to OFT are:
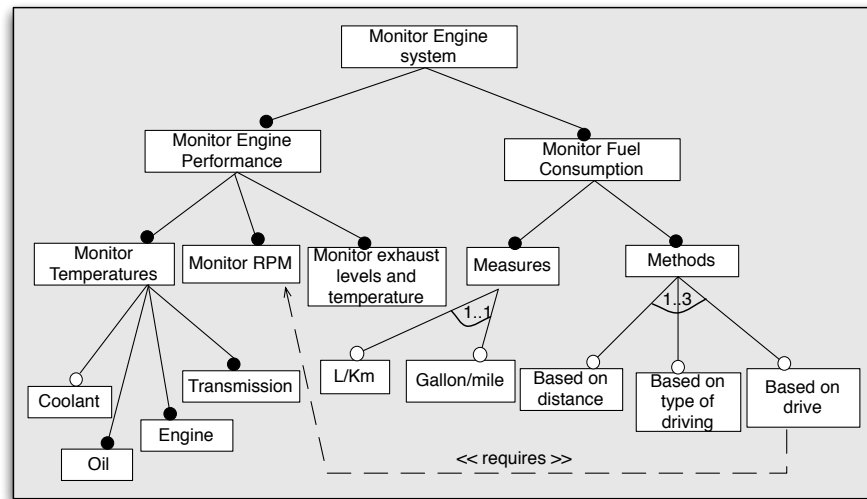
Figure 6.7: EFD: Monitor Engine System
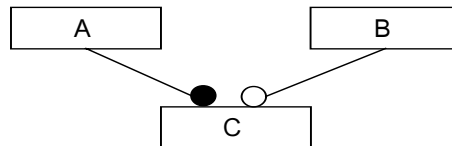
Figure 6.8: EFD: optional and mandatory edges (Riebisch et al., 2002)

1. The usual FD representation conventions are changed: the type of decomposition operator is not found in the decomposed feature anymore, nor on the departing edges but in the operand features: *single adaptors* (features with a circled 'S') represent an *xor*-decomposition of their father while *multiple adaptors* (features with a circled 'M') represent *or*-decomposition.

2. Mandatory features are decorated with a filled circle in contrast with hollow circle for optional features.

3. Graphical representations (dashed arrows) are given for the constraints *requires* and *mutex*.

4. No textual representation is given for constraints.

6.3.8 General Overview

The general overview on FD languages given in Figures 6.11 and 6.12 immediately highlights aesthetic differences among them. Particularly, we observe at least five different representations for the *xor*-decomposition (Figure 6.10). These issues mainly concern *concrete syntax*, i.e. what the users see. In this work, our main purpose is to study what is really behind the pictures, i.e.,
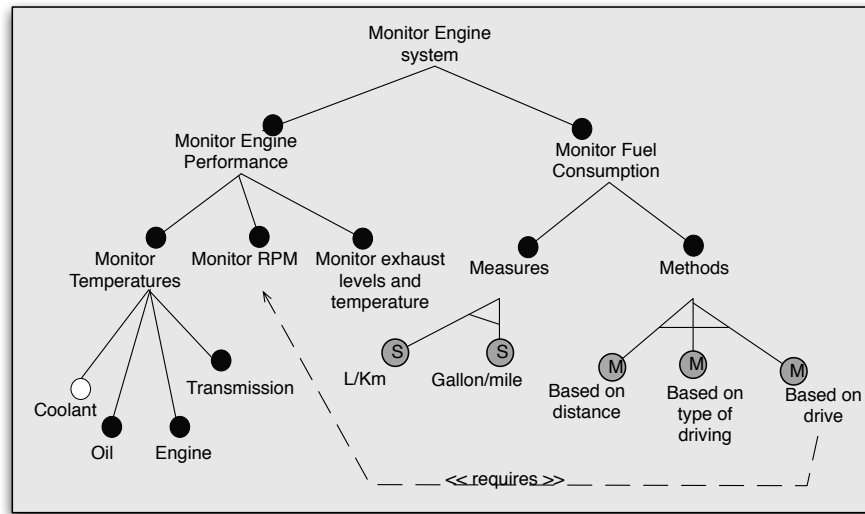
Figure 6.9: PFT: Monitor Engine System

semantics. This crucial issue has also been investigated by other authors. Their proposals are briefly introduced in the following section (Section 6.4). Some of them will be presented in detail in Chapter 9.
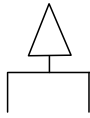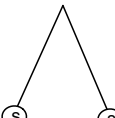
Figure 6.10: Concrete syntaxes for *xor*-decomposition

Main Characteristics	Example

Survey Short Name: OFT
Method: FODA
Author(s): Kang *et al.* 1990
Graph Type: Tree
Decomposition: *and, xor, opt*
Constraint Types: Textual
Formal Semantics:
    - Originally: none
    - A posteriori: Bontemps *et al.* 2004, Schobbens *et al.* 2006, Schobbens *et al.* 2007.	

Survey Short Name: OFD
Method: FORM
Author(s): Kang *et al.* 1998
Graph Type: DAG
Decomposition Types: *and, xor, opt*
Constraint Types: Textual
Formal Semantics:
    - Originally: none
    - A posteriori: Schobbens *et al.* 2006, Schobbens *et al.* 2007.	

Survey Short Name: RFD
Method: FeatuRSEB
Author(s): Griss *et al.* 1998
Graph Type: DAG
Decomposition Types: *and, xor, opt, or*
Constraint Types: Textual & Graphical
Formal Semantics:
    - Originally: none
    - A posteriori: Schobbens *et al.* 2006, Schobbens *et al.* 2007.	

Survey Short Name: GPFT
Method: Generative Programming
Author(s): Czarnecki *et al.* 2000
Graph Type: Tree
Decomposition Types: *and, xor, opt, or*
Constraint Types: Textual
Formal Semantics:
    - Originally: none
    - A posteriori: Batory 2005, Czarnecki *et al.* 2005b, Benavides *et al.* 2005a, Sun *et al.* 2005, Wang *et al.* 2005a, Schobbens *et al.* 2006, Schobbens *et al.* 2007.	

Figure 6.11: Survey of FD languages (1/2)

Main Characteristics	Example

Survey Short Name: VBFD
Method: /
Author(s): van Gurp *et al.* 2001
Graph Type: DAG
Decomposition Types: *and, xor, or, opt*
Constraint Types: Textual & Graphical
Formal Semantics:
    - Originally: none
    - A posteriori: Schobbens *et al.* 2006, Schobbens *et al.* 2007.	

Survey Short Name: EFD
Method: FORE
Author(s): Riebisch *et al.* 2002
Graph Type: DAG
Decomposition Types: *card, opt*
Constraint Types: Textual & Graphical
Formal Semantics:
    - Originally: none
    - A posteriori: Schobbens *et al.* 2006, Schobbens *et al.* 2007.	

Survey Short Name: PFT
Method: PLUSS
Author(s): Eriksson *et al.* 2005
Graph Type: Tree
Decomposition Types: *and, xor, opt, or*
Constraint Types: Graphical
Formal Semantics:
    - Originally: none
    - A posteriori: Schobbens *et al.* 2006, Schobbens *et al.* 2007.	

Figure 6.12: Survey of FD languages (2/2)

6.4 Formally Defined FD Languages

In this section, we survey and summarise the main proposals providing a formal definition for a FD language. We have identified the following proposals: (van Deursen and Klint, 2002; Mannion, 2002; Bontemps et al., 2004; Cechticky et al., 2004; Czarnecki et al., 2005c,b; Benavides et al., 2005a; Batory, 2005; Wang et al., 2005b; Sun et al., 2005; Asikainen et al., 2006; Janota and Kiniry, 2007). In the sequel, we successively present these proposals in a chronological order. For each proposal, we establish:

1. Which FD language is formally defined,

2. How this FD language is formally defined and

3. Which tool is provided to support this FD language and its semantics.

6.4.1 Van Deursen and klint

In (van Deursen and Klint, 2002), the authors define a textual FD language for which they formalise a semantics. The approach is original as it is the first time that a FD language has been formally defined. This FD language has the particularity to limit feature sharing to the leaves of the FD. The abstract syntax of the language is defined with a feature grammar. The semantic definition is based on a feature algebra with *rewriting rules* specified in the *ASF and SDF formalisms* (Brand et al., 2001). Then, the Meta-Environment (Brand et al., 2001) associated to these formalisms is used to generate tool support for the defined language. This language will be studied in detail in Section 9.3.

6.4.2 Mannion

In (Mannion, 2002), the author proposes an abstract syntax and a semantics for a product line model language that we called PLMD. This formal definition is the first one that (1) allows unrestricted feature sharing and (2) maps FDs to logical expressions. The abstract syntax is represented within a lattice that allows requirements (features) sharing. Later, a concrete syntax for this language has been proposed in the PLUSS method (Eriksson et al., 2005).

In the original version, the author does not use the terms "FD" or "FD language" however he defines the notion of product line model. A product line model is "a pool of numbered, atomic and natural-language requirements, a domain dictionary and a set of discriminants" (Mannion, 2002). These requirements are related through parent-child links. This definition is closely related to FDs, if we consider that requirements are replaced by features. The semantics of the language is defined by translations from product line models to logical expressions. Basically, requirements are mapped to atoms and relationships between requirements are mapped to logical expressions. These product line models are then validated by instantiating the resulting logical expression and by using propositional calculus.

6.4.3 Bontemps *et al.*

In (Bontemps et al., 2004; Schobbens et al., 2007), we have formally defined a new FD language called VFD that is a simplified form of EFD while other FD languages are extensions of the original

one (OFD). The main ideas are that VFD forms a DAG and that *card*-decomposition is the only type of feature decomposition allowed. All the constraints and types of features present in other concrete syntaxes are mapped to *card* Boolean operators. The VFD abstract syntax is therefore minimal and, as we will see further, sufficient to represent every possible product line. Initially, the VFD semantics has been used to define the semantics of our family of FD languages, FFD (Chapter 8). In this work, the VFD semantics is defined as a translation to FFD. As mentioned before, concrete syntaxes are not our major concern. However, as illustrated in Figure 6.13, we suggest, for the VFD concrete syntax, to keep arrows for requires and excludes constraints and to represent optional features with hollow circles to facilitate FD visualisation.
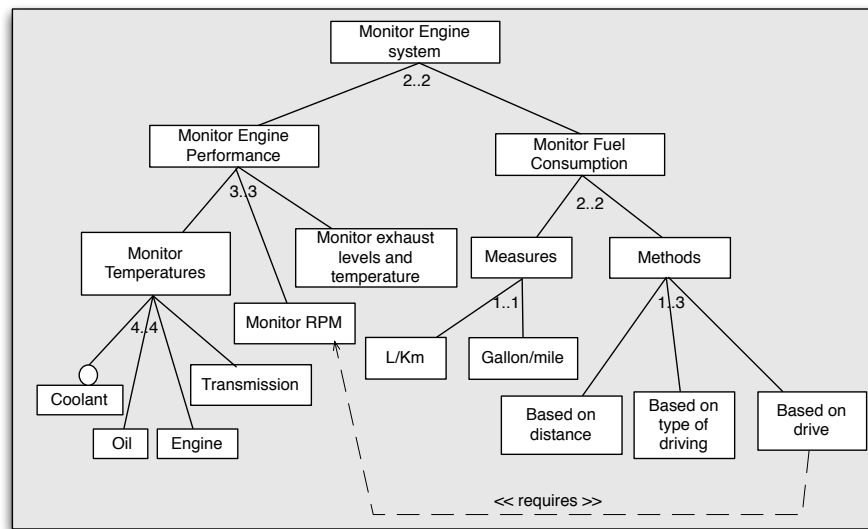
Figure 6.13: VFD: Monitor Engine System

6.4.4   Cechticky *et al.*

In (Cechticky et al., 2004), the authors propose a meta-model and a semantics for an XML-Based FD language that we call XFD. The originality of their approach is that the FD is expressed as an XML document based on the defined meta-model. Similarly, the global composition constraints (requires, mutex) are described in another XML document called constraint family model. The notion of *feature macro* is introduced to split a large diagram into smaller modules. The semantic definition is based on translations from FDs to *composition rules* specified in the *eXtensible Stylesheet Language (XSL)*. These composition rules are then evaluated on XFD. This approach has been implemented within the XFeature Tool (Rohlik and Pasetti, 2006).

6.4.5 Czarnecki *et al.*

In (Czarnecki et al., 2005b), the authors define a new FT language based on (Eisenecker and Czarnecki, 2000) and its semantics to account for staged configuration. They introduce *feature cardinalities* (the number of times a feature can be repeated in a product) in addition to the more usual (*group*) cardinality. Foremost, a semantic domain is proposed where the full shape of the unordered tree is important, including repetition and decomposable features. The semantics is defined in a 4-stage process where FDs are translated in turn into an extended abstract syntax, a context-free grammar and an algebra. In (Czarnecki et al., 2005c), the authors provide an even richer syntax. The semantics of the latter is yet to be defined, but is intended to be similar to (Czarnecki et al., 2005b). This semantics has been implemented in the Eclipse Feature Modelling Plug-in (fmp) (Czarnecki et al., 2005a) and is based on *Binary Decision Diagrams (BDD)* algorithms (Brace et al., 1990).

6.4.6 Benavides *et al.*

In (Benavides et al., 2005a), the authors apply the techniques of constraint programming to provide a semantics for FTs and to reason on them. They extend the FT language proposed in (Eisenecker and Czarnecki, 2000) with extra-functional features, attributes and relations between attributes. Subsequently, they describe tool support based on mappings between these FTs and *Constraint Satisfaction Problems (CSP)* (Tsang, 1995). This semantics has been implemented in the tool FAMA (Benavides et al., 2007).

6.4.7 Batory

In (Batory, 2005), the author provides a semantics to the FT language proposed in (Eisenecker and Czarnecki, 2000). This semantic definition is based on translations from FTs to both *iterative tree grammars* and *propositional logic*. This FT language uses the whole propositional logic as constraint language. The semantic definition is supported by the guidsl tool (Batory, 2005) and enables (1) to propagate constraints using off-the-shelf Logic Truth Maintenance Systems (LTMS) (Forbus and de Kleer, 1993) and (2) to check feature compatibility and constraints using SAT-Solvers. This language will be studied in detail in Section 9.4.

6.4.8 Sun *et al.*

In (Sun et al., 2005), the authors first provide a semantics for the FT language proposed in (Eisenecker and Czarnecki, 2000) based formal specifications respecting the Z language (Spivey, 1988; Woodcock, 1993; Saaltink, 1997). Secondly, they validate the correctness of this semantics using the Z/EVES (Saaltink, 1997) theorem prover. Finally, they implement this semantics in Alloy (Jackson, 2006) to check feature model consistency, resolvability and equivalence.

6.4.9 Wang *et al.*

In (Wang et al., 2005a), the authors define a semantics for the FT language proposed in (Eisenecker and Czarnecki, 2000) using ontologies. A semantic web environment is used to model and verify FDs with *OWL DL* (McGuinness and van Harmelen, 2004). Features are represented as

OWL classes and feature constraints as OWL properties. OWL reasoning tools (Renamed ABox and Concept Expression Reasoner (RACER)) (Haarslev and Möller, 2003) allow checking fully automatically inconsistencies. The parallel between feature modelling and semantic web is interesting as both represent and structure concepts in a particular domain.

6.4.10   Asikainen *et al.*

In (Asikainen et al., 2006), the authors propose to synthesise the conceptual foundation of feature models by introducing a domain ontology for feature modelling, called "Forfamel". They suggest a new FD language based on Forfamel that we name FoFD. The semantics of FoFD is defined via a translation from the Forfamel ontology to a general purpose knowledge representation language called Weight Constraint Rule Language (WCRL) (Soininen et al., 2001). FoFD and its semantics are supported by a tool called Kumbang Configurator. It helps to configure the feature model; it allows checking whether a configuration is valid and, if not, it identifies the constraints that are not satisfied.

6.4.11   Janota and Kiniry

In (Janota and Kiniry, 2007), the authors propose a meta-model and a semantics for the FT language proposed in (Eisenecker and Czarnecki, 2000) extended with feature attributes, cardinalities and cloning. Its semantic domain is defined as a set of sets of valid configurations. The semantic function relies on a restriction function that indicates whether a configuration is valid or not. This approach is similar to our. However, their semantics is based on high-order logic while our and others on propositional logic. In this case, we don't see any reason to prefer high-order logic. We suspect that the tool implementing their semantics has been selected beforehand. This tool is based on high-order logic and is called Prototype Verification System (PVS) (Owre et al., 1992).

6.4.12   General Overview

The general overview and comparison proposed in Table 6.2 underlines several interesting observations. Firstly, we notice a profusion of languages, semantics, formalisms and tools. Secondly, the semantics of these FD languages are often defined by translations to other languages that have an already well-defined semantics. However, the semantic domains of these languages are not necessarily suitable for FD languages. Most of the time, the first intention of the authors was to provide efficient tool support for FD languages, directly based on these translations.

The comparison presented in Table 6.2 still remains superficial. Well-defined criteria should be applied to compare formal FD languages following a clear and systematic method. We have experienced that these issues are even more crucial when FD languages with different semantics and formalisms are compared.

Author(s)	FD Language	Mapped to	Tool Support
van Deursen and Klint (2002)	vDFD	ASF + SDF	Meta-Environment
Mannion (2002)	PLMD	Propositional Logic	
Bontemps et al. (2004)	VFD	Propositional Logic	VFD reasoning tool
Cechticky et al. (2004)	XFD	XSL	XFeature
Czarnecki et al. (2005b)	GPFT + Cardinalities	BDD	fmp
Benavides et al. (2005a)	GPFT + Attributes	CSP	FAMA
Batory (2005)	GPFT + Propositional Logic	Propositional Logic	guidsl: LTMS + SAT
Sun et al. (2005)	GPFT	Z	Alloy
Wang et al. (2005a)	GPFT	OWL	RACER
Asikainen et al. (2006)	FoFD	WCRL	Kumbang-Configurator
Janota and Kiniry (2007)	GPFT + Cardinalities + Attributes and Cloning	High-order Logic	PVS

Table 6.2: Formal FD Languages

6.5   Chapter Summary

Throughout this chapter, we have surveyed FD languages. First, we have described the graphical aspects and particularities of informal FD languages. Then, we have given an overview of the main proposals that formally define FD languages. The current situation is characterised by a profusion of languages and semantics, many loosely defined, and loosely compared with their "competitors". Clearly defined criteria and terminology are paramount to structure the research efforts that focus on providing effective modelling techniques. The research on feature modelling and FD languages should be no exception. Therefore, in the next Chapter 7, we propose a method to evaluate FD languages according to formal criteria previously defined in Chapter 5.

Chapter 7

FD Languages: A Comparison Method

In Chapter 3, we have presented the basic concepts and goals of the SEQUAL Framework. We have also extended this framework with formal properties to improve language and model evaluation. In Chapter 6, we have surveyed formal and informal FD languages. In the present chapter, we suggest a method, previously published in (Schobbens et al., 2006), to compare FD languages from a formal perspective. In the next chapter, we will apply this method to the surveyed FD languages.

Our investigation targets FD languages rather than the FDs themselves. A problem with evaluating model (diagram) quality is that representative objects of study – that are models – do not always exist, or at least are not easily available. Indeed, this is the case for FDs which (1) are an emerging modelling paradigm, and (2) have the purpose of representing highly strategic company information. Therefore, representative FDs are almost nowhere to find. At this stage, we thus thought we should concentrate on improving the appropriateness of FD *languages* before any standardisation is attempted and they hopefully become widespread in industry. Not being able to assess FD qualities directly, our investigations are rather targeted on *FD language quality*. The idea is to investigate how formal language properties may improve FD language quality as defined in (Krogstie, 2001b; Krogstie et al., 2006).

In Chapter 5 we have proposed to study four formal quality criteria to compare FD languages: Computational Complexity, Expressiveness, Embeddability and Succinctness. Once decision problems have been identified, *Computational Complexity* is applicable to any formal language without any adaptation. Although formality is required, comparing languages wrt complexity does not require that these languages follow Harel and Rumpe's principles (Chapter 4) or share the same semantic domain. Nevertheless, these conditions are necessary to compare FD languages according to *Expressiveness, Embeddability* and *Succinctness*. Formally it means that (1) for each language X_i to be compared, we exactly know their syntactic domain $\mathcal{L}_{X_i}$, semantic domain $\mathcal{S}_{X_i}$ and semantic function $\mathcal{M}_{X_i}$ and (2) $\mathcal{S}_{X_1} = \mathcal{S}_{X_2} = \ldots = \mathcal{S}_{X_i} = \ldots = \mathcal{S}_{X_n}$. This ideal situation almost never occurs in practice. Most of the time, we have to cope with three different cases:

- Case 1: FD Languages that are described in natural language *without formal semantics* at all (Section 7.3). Proposals of this kind can be found in (Kang et al., 1998; Griss et al., 1998; Eisenecker and Czarnecki, 2000; Riebisch et al., 2002; Riebisch, 2003; van Gurp et al., 2001;

Eriksson et al., 2005);

- Case 2: FD Languages *with a formal semantics* but defined with quite *different principles* from what is advocated in Chapter 4 (Section 7.4.1). Proposals of this kind can be found in (Mannion, 2002; Czarnecki et al., 2005c,b; Batory, 2005; Sun et al., 2005; Wang et al., 2005a; Benavides et al., 2005a; Asikainen et al., 2006);

- Case 3: FD Languages *with a formal semantics* compliant with what is advocated in Chapter 4 but using *different semantic domains* (Section 7.4.2). Proposals of this kind can be found in (van Deursen and Klint, 2002)[*] or appear when FD languages from Case 2 are reformulated.

The structure of this chapter is as follows. First, we list and define in Section 7.1 essential decision problems that allow comparing FD languages according to Computational Complexity. Then a comparison process to evaluate FD languages according to Expressiveness, Embeddability and Succinctness is proposed in Section 7.2. Finally, we describe how informal and formal languages should be adapted in order to compare them in Sections 7.3 and 7.4, respectively.

7.1 Computational Complexity

Computational complexity analysis is important because its results help the language engineer to evaluate the scalability of the tool support for their language. Formalisation of both the language syntax and semantics is a necessary prerequisite to devise precise questions over the language that the tool can answer with an efficiency that must be evaluated. Complexity results then give an indication about the worst case and how to handle it.

Evaluating FD languages according to computational complexity criterion requires to identify decision problems (questions) that check automatically essential properties on FDs (a.k.a. automated analysis (Batory et al., 2006)). Once they have been formally stated, their complexity can be studied. Here, we formally define six essential decision problems associated to FD languages, or more generally any language whose semantic domain is a set. These decision problems are: Satisfiability (Definition 7.1.1), Product-Checking (Definition 7.1.2), Equivalence (Definition 7.1.3), Intersection (Definition 7.1.4), Inclusion (Definition 7.1.5) and Union (Definition 7.1.6).

7.1.1 Satisfiability

Satisfiability is a fundamental property. For any logic, satisfiability is the problem of determining if the variables of a given formula can be assigned to make the formula evaluate to TRUE. In FDs, satisfiability is the problem of determining if a FD represents a product line with at least one product. It must be checked for the product line and also for the intermediate FDs produced during a staged configuration (Czarnecki et al., 2005c).

Definition 7.1.1 (Satisfiability) *A FD d is* satisfiable *iff* $\mathcal{M}[\![d]\!] \neq \emptyset$.

---

[*]van Deursen and Klint language definition does not follow exactly the principles and terminology presented in Chapter 4, however its syntactic domain, semantic domain and semantic function can be deduced from the original definition with minor adaptations.

7.1.2 Product-Checking

Product-checking (a.k.a model-checking) verifies whether a given product (set of primitive features) is in the product line of a given FD or not.

Definition 7.1.2 (Product-Checking) *A product c is in the product line of a FD d iff $c \in \mathcal{M}[\![d]\!]$.*

7.1.3 Equivalence

Equivalence of two FDs is useful whenever we want to compare two versions of a product line (for instance, after a refactoring). A related problem is to produce a product showing their difference when they are not equivalent.

Definition 7.1.3 (Equivalence) *Two FDs d_1 and d_2 are* equivalent *iff $\mathcal{M}[\![d_1]\!] = \mathcal{M}[\![d_2]\!]$.*

7.1.4 Intersection

Intersection is useful when two feature interference engineers work independently and therefore obtain two different restrictions of the initial product line. To put their work together, one must produce a FD representing the intersection.

Definition 7.1.4 (Intersection) *Given two FDs d_1 and d_2, a FD d_3 represents the* intersection *of d_1 and d_2 iff $\mathcal{M}[\![d_3]\!] = \mathcal{M}[\![d_1]\!] \bigcap \mathcal{M}[\![d_2]\!]$.*

7.1.5 Inclusion

Inclusion is useful to check whether a product line is included into another one. For instance, it checks whether a new product line takes into account all the products contained in the previous one.

Definition 7.1.5 (Inclusion) *Given two FDs d_1 and d_2, d_1 is included into d_2 iff $\mathcal{M}[\![d_1]\!] \subseteq \mathcal{M}[\![d_2]\!]$.*

7.1.6 Union

Union is useful when teams validate in parallel the feature combinations that lead to an acceptable product, without feature interference. Their work can be recorded in separate FDs. The union of these FDs will represent the validated products.

Definition 7.1.6 (Union) *Given two FDs d_1 and d_2, a FD d_3 represents the* union *of d_1 and d_2 iff $\mathcal{M}[\![d_3]\!] = \mathcal{M}[\![d_1]\!] \bigcup \mathcal{M}[\![d_2]\!]$.*

7.2   Comparison Process

Before comparing FD languages according to expressiveness, embeddability and succinctness, we first need to make them suitable for comparison. Hence, the three cases mentioned in the introduction of this chapter should be handled. For this purpose we propose a comparison process for FD languages such as illustrated in Figure 7.1. Let us call X the language we want to compare with other languages respectively $Y_1, ..., Y_n$ which, we assume, (1) are formalised according to (Harel and Rumpe, 2004) and (2) share the same semantic domain.

Both formal and informal languages receive a specific attention when they need to be compared. However, the distinction between formal and informal languages is not always clear. In section 6.1, we have clarified this distinction, following (Harel and Rumpe, 2004) where a language is considered formal if it possesses mathematical definitions for its syntax and semantics. Once the studied FD language has been qualified as formal or informal, the necessary properties to compare languages can be studied and the FD languages adapted in consequence.

The activity diagram (OMG, 2008) of Figure 7.1 illustrates how we propose to adapt and compare FD languages. The corresponding activities will be described in the next sections. The succession of activities is determined according to six different choices (represented as white diamonds):

- The first one (X is formal?) concerns the distinction between formal and informal FD languages which both receive specific adaptations.

- The second one (X in FFD?) concerns the formalisation of informal languages. Such languages could be totally redefined following the principles recalled in Chapter 4. To facilitate this, we proposed a configurable formal definition named FFD (Chapter 8). FFD deliberately covers OFT (Kang et al., 1990), OFD (Kang et al., 1998), RFD (Griss et al., 1998), GPFT (Eisenecker and Czarnecki, 2000), EFD (Riebisch et al., 2002), VBFD (van Gurp et al., 2001), PFT (Eriksson et al., 2005) and VFD (Bontemps et al., 2004). The activity "Configure FFD" will be detailed in Chapter 8. We can already underline that this activity highly facilitates the adaptations of many informal languages. Indeed, FFD is based on Harel and Rumpe's principles (H&R) (Chapter 4) and provides FD languages with a common semantic domain. Hence, these FD languages are automatically adapted and ready for comparison.

- The third one (X follows H&R?) concerns how formal languages are defined. The question is: Does the studied language follow Harel and Rumpe's principles (H&R) (Chapter 4) or not? If not the language definition should be adapted.

- The fourth one ($\mathcal{S}_X = \mathcal{S}_{Y_i}$?) concerns the semantic domains. When both languages share the same semantic domain, they can be directly compared. Otherwise, their semantic domains should be related by abstraction functions.

- The fifth one (X is as expressive as Y_i?) concerns the expressiveness analysis results. Indeed, if two languages do not have the same expressiveness there is no way to find a correct embedding between both languages. Hence, embeddability and succinctness analyses are irrelevant.
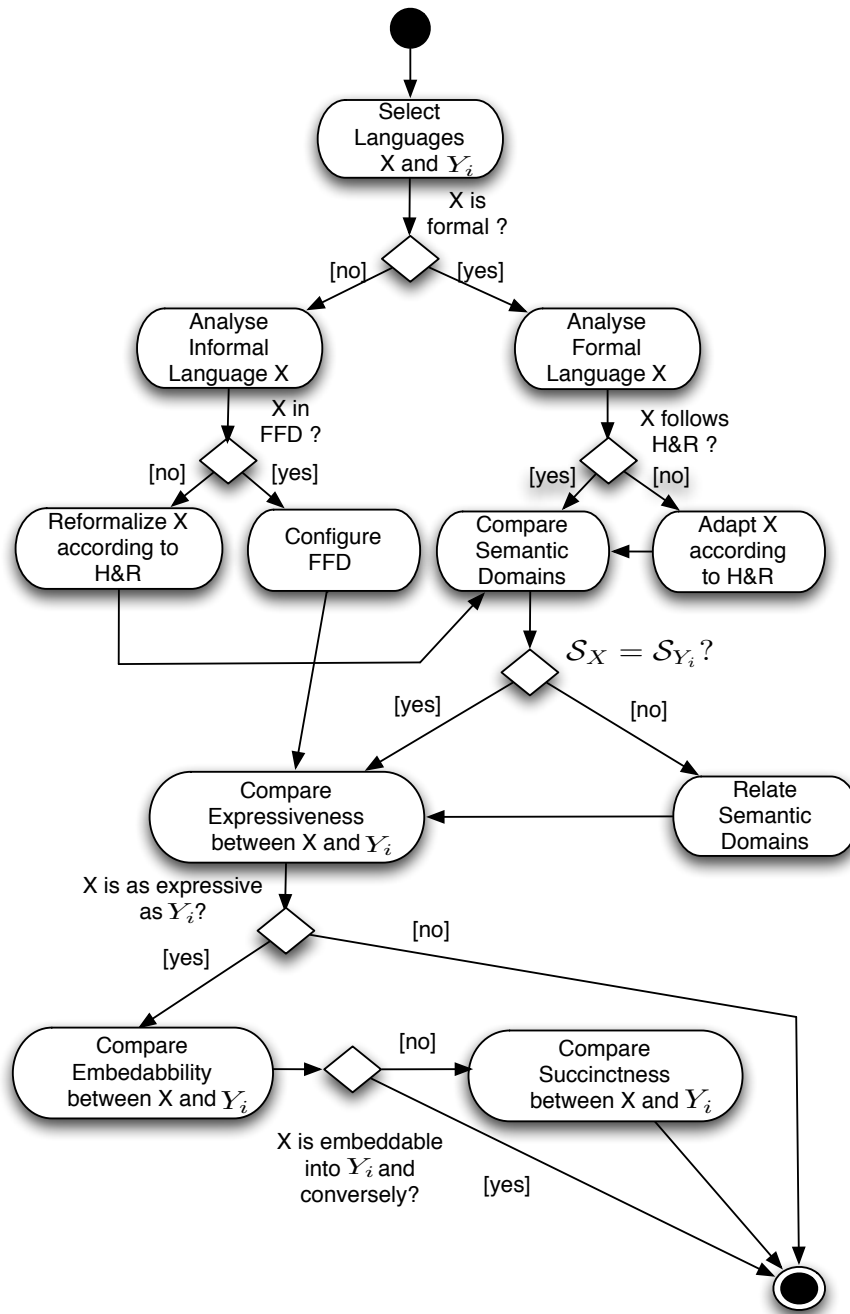
Figure 7.1: Comparison Method for FD languages

- The sixth one (X is embeddable into Y_i and conversely?) concerns the embeddability analysis results. Indeed if a language (X) is embeddable into another language (Y_i), then the translation between X and Y_i is linear and therefore X is linearly as succinct as Y_i. Conversely, if Y_i is embeddable into X then the translation between Y_i and X is also linear and Y_i is linearly as succinct as X. Otherwise, when one non linear translation exists in one way or another, succinctness should be studied.

7.3   Informal FD languages

When *X* has no formal syntax or semantics (Case 1), two alternatives exist:

- The first alternative is to formalise the concerned FD language from scratch according to Harel and Rumpe's principles (Chapter 4). Then, *X* is completely defined independently from $Y_1, ..., Y_n$.

- The second alternative is to make scale economies and to reuse the configurable formal definition, called FFD, that we propose and which defines several FD languages at once. In Chapter 6, we observed that most of the FD languages largely share the same goals, the same constructs and, as we understood from the informal definitions, the same (FODA-inspired) semantics. For this reason, we propose to define not one FD language but a family of related FD languages (Chapter 8). We suggest a parametric and configurable definition called FFD, in which parameters correspond to variations in the FD language's abstract syntaxes: $\mathcal{L}_X, \mathcal{L}_{Y_1}, \dots, \mathcal{L}_{Y_n}$. Once these parameters are instantiated, FFD automatically provides an appropriate formal definition for the desired FD language. FFD follows and slightly adapts the principles advocated in Chapter 4. The semantic domain and semantic function are common to all FD language variants, maximising semantic reusability.

  With this FFD definition, we are confined to handle FD languages whose only significant variations are in their abstract syntax. For FD languages with different semantic choices (Section 8.4), FFD is not sufficient. For instance, we present in Section 7.4.2 a method to compare FD languages with different semantic domains.

7.4   Formal FD languages

When *X* has formal syntax and semantics, two alternatives exist:

1. The first alternative concerns FD languages with a formal semantics defined with different principles from what is advocated in Chapter 4 (Case 2). The adaptation consists in improving the language's semantic definition according to these principles (Section 7.4.1).

2. The second alternative concerns FD languages with a formal semantics compliant with what is advocated in Chapter 4 but using different semantic domains (Case 3). The adaptation consists in relating the language's semantic domains (Section 7.4.2).

7.4.1 Adapt Semantic Definition to Harel and Rumpe's Principles

The second case is when a language X actually has a formal semantics, but expressed in a form that does not comply with the one advocated by Harel and Rumpe (Chapter 4) (Case 2). To facilitate the assessment of the criteria we are interested in, it is convenient to reformulate the way in which the FD languages to be compared are defined. This reformulation is due to missing mathematical definitions of $\mathcal{L}_X$, $\mathcal{S}_X$ and $\mathcal{M}_X$. Therefore, $\mathcal{L}_X$, $\mathcal{S}_X$ and $\mathcal{M}_X$ need to be clarified. Typically, $\mathcal{L}_X$ is clear but $\mathcal{S}_X$ and $\mathcal{M}_X$ are not. Most of the time, the semantics of X is given by describing a transformation of X's diagrams to another language, say W, which is formal. W does not even need, and usually it is not, to be a FD language. Therefore, the semantic domain might be very different from the one intuitively thought of for FDs. The main motivation for formalising this way is usually because W is supported by tools. The problem is that these kinds of "indirect", or tool-based, semantics complicate the assessment of the language[†].

The main advantage of adapting a formal language according to Harel and Rumpe's principles (Chapter 4) rather than completely reformalising an informal language is that formalisation decisions are usually much more straightforward since they have already been made. However, they might be hard to dig out if they are coded in a tool.

7.4.2 Relate Semantic Domains

The third case is when we have a clear and self-contained mathematical definition of $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{M}$ (either from the origin, or having previously gone through Case 1 or 2) but the semantic domains of the languages to be compared differ (Case 3). Hence, they cannot be compared directly for expressiveness, embeddability and succinctness. They all rely on the hypothesis that the languages under study share a *common semantic domain*. But, in general, the semantic domain of the languages may differ. Therefore, X has a formal semantics with clear $\mathcal{L}_X$, $\mathcal{S}_X$ and $\mathcal{M}_X$ but $\mathcal{S}_X \neq \mathcal{S}_{Y_i} (i \in \{1, ..., n\})$. In this case, we thus need to define a relation between the semantic domains.

Comparing languages with different semantic domains is actually possible, but it requires preliminary work (de Boer and Palamidessi, 1994). Consider two languages with their *syntactic domains* $\mathcal{L}_1$ and $\mathcal{L}_2$ and two different *semantic domains*, respectively $\mathcal{S}_1$ and $\mathcal{S}_2$. Their *semantic functions* are respectively $\mathcal{M}_1$ and $\mathcal{M}_2$. We must first compare intuitively the two domains to determine the information they share. We then create a domain $\mathcal{S}$ for this shared information and provide functions $\mathcal{A}_1 : \mathcal{S}_1 \rightarrow \mathcal{S}$ and $\mathcal{A}_2 : \mathcal{S}_2 \rightarrow \mathcal{S}$, called *abstractions*. The purpose of these abstraction functions is to remove additional information and keep the "core" of the *semantic domain*, where we will perform the comparisons. However, the question of the relevance of this discarded information remains and should be studied carefully.

A simple but frequent case is illustrated in Figure 7.2, where domain $\mathcal{S}_1$ is too rich and contains more information than $\mathcal{S}_2$; we then take $\mathcal{S}_2$ as the common domain. An abstraction function $\mathcal{A}$ removes from elements of $\mathcal{S}_1$ their supplementary information and maps them into $\mathcal{S}_2$. It then makes sense to look for two translations between their syntactic domains: $\mathcal{T} : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ and $\mathcal{T}' : \mathcal{L}_2 \rightarrow$

[†]Even more if W is also given a formal semantics in a similarly "indirect" way, just as X.
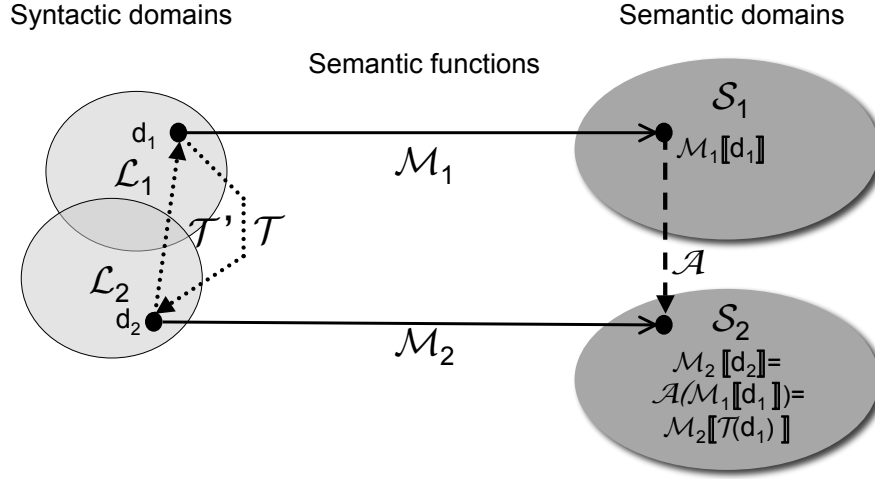
Figure 7.2: Abstracting a semantic domain

$\mathcal{L}_1$. They are translations for the abstracted semantics $\mathcal{A} \circ \mathcal{M}_1$, and can thus be used to compare languages for expressiveness, embeddability and succinctness. In (de Boer and Palamidessi, 1994), the authors mainly study the translation $\mathcal{T}'$ indicating if and how the diagrams of the language with the simplest semantic domain ($\mathcal{S}_2$) are translatable into the language with the more complex semantic domain ($\mathcal{S}_1$). This translation $\mathcal{T}'$ is often more complex. Here, the translation $\mathcal{T}$ is also interesting since we consider FFD as the family of FD languages with the simplest semantic domain. Our goal is to study whether every FD language can be easily translated into FFD while preserving their original abstracted semantics. This is what we call semantic equivalence relatively to the abstraction function $\mathcal{A}$.

Hence, if we apply $\mathcal{T}$ to d_1 in the syntactic domain $\mathcal{L}_1$ we will obtain d_2 in the syntactic domain $\mathcal{L}_2$ with the same abstracted semantics. Semantically, if we apply the semantic function $\mathcal{M}_1$ to d_1 and then $\mathcal{A}$ or, if we apply $\mathcal{T}$ to d_1 and then $\mathcal{M}_2$, we will map to the same element in $\mathcal{S}_2$: $\forall d_1 \in \mathcal{L}_1 : \mathcal{A}(\mathcal{M}_1[\![d_1]\!]) = \mathcal{M}_2[\![\mathcal{T}(d_1)]\!]$.

Similarly, if we apply $\mathcal{T}'$ to d_2 in the syntactic domain $\mathcal{L}_2$ we will obtain d_1 in the syntactic domain $\mathcal{L}_1$ with the same abstracted semantics. Semantically, if we apply the semantic function $\mathcal{M}_2$ to d_2 or, if we apply $\mathcal{T}'$ to d_2 then $\mathcal{M}_1$ and finally $\mathcal{A}$, we will map to the same element in $\mathcal{S}_2$: $\forall d_2 \in \mathcal{L}_2 : \mathcal{M}_2[\![d_2]\!] = \mathcal{A}(\mathcal{M}_1[\![\mathcal{T}'(d_2)]\!])$.

When applied to more than two languages, this method will create many semantic domains related by abstraction functions. These abstraction functions can be composed and will describe a category of semantic domains (Figure 7.3). At the syntactic level, the translations can also be composed to yield embeddability and succinctness results. Similarly, the composition of embeddings yields an embedding. In Figure 7.3, we consider FD languages with six different syntactic domains ($\mathcal{L}_1$, ..., $\mathcal{L}_6$) and six semantic domains ($\mathcal{S}_1$, ..., $\mathcal{S}_6$). These languages do not share the same semantic domain except for $\mathcal{L}_4$ and $\mathcal{L}_6$ where $\mathcal{S}_4 = \mathcal{S}_6$. $\mathcal{S}_5$ is considered as the more abstract

or common semantic domain. It can be reached by composition of the abstraction functions ($\mathcal{A}_1$, ..., $\mathcal{A}_6$). Conversely, $\mathcal{S}_1$ is considered as the less abstract semantic domain.
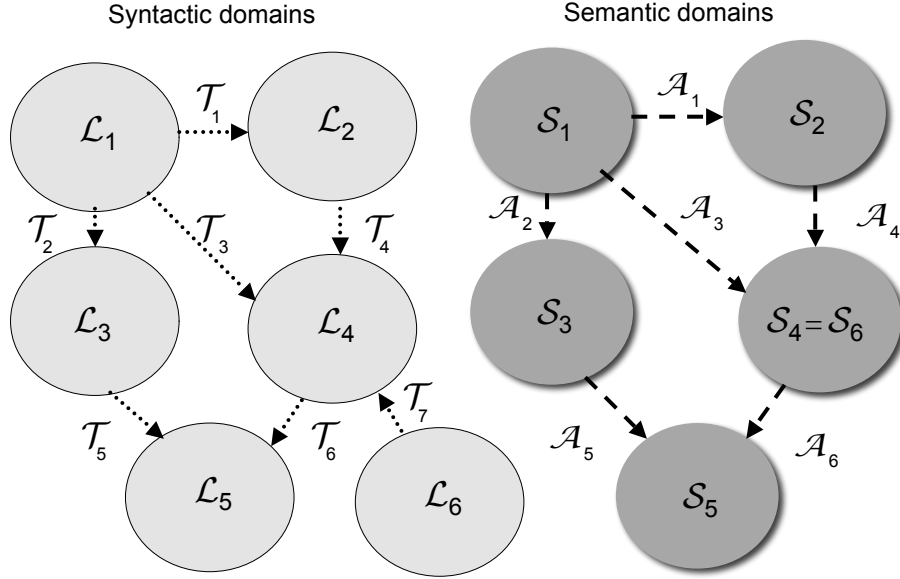
Figure 7.3: Semantic Domain Category

When two languages share a common semantic domain and when a translation exists between their syntactic domains, their results concerning expressiveness are identical. Hence, we need to go further and to study the translation properties that determine embeddability and succinctness. This is the case for $\mathcal{L}_4$ and $\mathcal{L}_6$ in Figure 7.3 which share the same semantic domain and are related by a translation $\mathcal{T}_7$.

The situation is similar when two languages are amenable to a common semantic domain and when a translation exists between their syntactic domains. When two languages do not share the same semantic domain, their semantic domains should be related by an abstraction function in order to define a correct embedding between languages at the syntactic level. This is the case for $\mathcal{L}_1$ and $\mathcal{L}_2$ in Figure 7.3. The idea is that both semantic domains ($\mathcal{S}_1$ and $\mathcal{S}_2$) could be abstracted to one common semantic domain ($\mathcal{S}_5$). When a semantic domain category exists, the task is to construct an abstraction function between the given semantic domain and one member of the category. Once this abstraction function is defined, the semantic domain could be related to the common one by composition of abstraction functions. For instance, in Figure 7.3, several possibilities exist to relate $\mathcal{S}_1$ and $\mathcal{S}_5$. The abstraction function between S_1 and S_5 may result from three different compositions of already existing abstraction functions: $\mathcal{A}_5 \circ \mathcal{A}_2$ or $\mathcal{A}_6 \circ \mathcal{A}_3$ or $\mathcal{A}_6 \circ \mathcal{A}_4 \circ \mathcal{A}_1$.

7.5   Chapter Summary

In this chapter, we have proposed a method that describes (1) how the formal criteria defined in Chapter 5 should be used to compare FD languages, (2) which properties these languages should have to be comparable and (3) how to adapt both formal and informal FD languages to satisfy these properties. In Chapter 8, we further equip our comparison method with FFD, a configurable definition for a family of FD languages. In Chapter 9, we will apply our method to compare both informal FD languages covered in FFD and selected formal FD languages.

Chapter 8

FFD: a Formal Configurable Definition

In Chapter 4, we have recalled the principles defined by Harel and Rumpe (Harel and Rumpe, 2004) to improve the formal definition of languages. In the present chapter, we show how to define a family of FD languages according to these principles. This definition is original in the sense that it is configurable. It allows formally (re)defining most of the FD languages presented in Chapter 6. The semantics formally defined here, reflects how we understand the informal semantics of OFD described in FORM (Kang et al., 1998). This formal definition will be the basis for further reasoning on FD languages and for tool implementation.

The structure of this chapter is as follows. First, in Section 8.1, we formally define OFD (Kang et al., 1998) applying the principles proposed in Chapter 4. This first main contribution has been previously published in (Bontemps et al., 2005). Based on this OFD definition, we formally define in Section 8.2 a family of FD languages named Free Feature Diagram Language (FFD). This second main contribution has been previously published in (Schobbens et al., 2006). FFD definition still follows the principles described in Chapter 4 but with some necessary adaptations presented in Section 8.2.1. In Section 8.2.2, we apply these adapted principles to provide a formal and configurable definition of FFD. In Section 8.3, we illustrate how the FFD definition can be instantiated (parametrised) and, by this mean, we show how each member of the family can be formally defined with minimal efforts. Finally, we discuss in Section 8.4 the semantic choices that have been necessary to define FFD semantics.

8.1 OFD Definition

OFD (Kang et al., 1998) is the first extension of the seminal FD language (OFT) (Kang et al., 1990) proposed eight years later by the same authors. Both languages are frequently cited and experimented in practice within numerous product line case studies. As mentioned in Section 6.3.2, two variants of OFD exist. Only the simplest one will be formally defined here. This version simply extends OFT with DAGs instead of being limited to trees.

In Section 8.1.1, we formally define the OFD syntactic domain ($\mathcal{L}_{OFD}$). Then, in Section 8.1.2, we formally define the OFD semantic domain ($\mathcal{S}_{OFD}$). Finally, in Section 8.1.3, we formally define the OFD semantic function ($\mathcal{M}_{OFD} : \mathcal{L}_{OFD} \rightarrow PL$).

8.1.1   The syntactic domain of OFD ($\mathcal{L}_{OFD}$)

From the *concrete syntax* point of view, OFDs are graphical combinations of elementary symbols such as boxes (features), strings (feature names and textual constraints), lines (feature decomposition) and circles (on top of optional features). Defining the allowed combinations of these symbols involves describing where they should be placed, which size and colour they should have, etc. This is not necessarily difficult, but bulky.

The *abstract syntax* or *syntactic domain* defines the allowed essential syntactic structures behind these visualisation details. As mentioned earlier, for visual languages, the two most widespread ways to define an abstract syntax are: (1) *mathematical notation* (set theory) and (2) *meta-modelling*. Several meta-models of FDs exist in the literature (Fey et al., 2002b; Czarnecki et al., 2004; Cechticky et al., 2004; Czarnecki et al., 2005b,c; Chen et al., 2005; Benavides et al., 2005b; Asikainen et al., 2006; Djebbi and Salinesi, 2006). Nevertheless, we prefer the mathematical format for its greater universality, unambiguity, conciseness and suitability to undergo rigorous proofs.

A mathematical abstract syntax for OFD is given in Definition 8.1.1 and illustrated in Figure 8.1. The left part of this figure corresponds to one FD illustrating the concrete syntax of OFD. The right part illustrates the abstract syntax we have defined for OFD ($\mathcal{L}_{OFD}$). Please note that the graphical representation given for the abstract syntax is by no means a proposal for a new concrete syntax. It just serves to illustrate the formal definition.
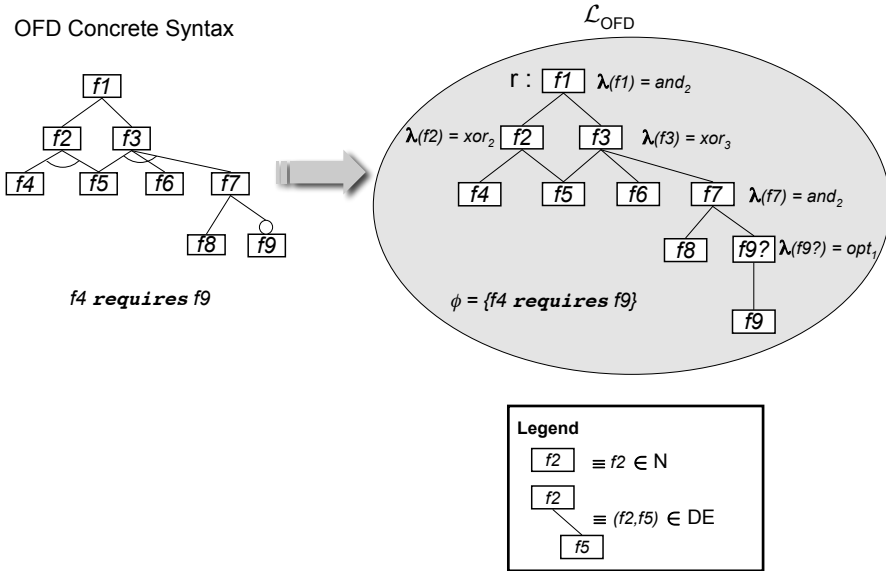

Figure 8.1: OFD's syntactic domain: $\mathcal{L}_{OFD}$

The illustrated FD is composed of nine features; one being optional (f_9), four being mandatory (f_1, f_2, f_3, f_8) and four being alternative ones (f_4, f_5, f_6, f_7). In terms of feature decomposition,

this FD contains one *and*-decomposition (f_1 into *and*(f_2, f_3)) and three *xor*-decompositions (f_2 into *xor*(f_4, f_5), f_3 into *xor*(f_5, f_6, f_7) and f_7 into *and*(f_8, f_9)). As the feature f_5 is shared by f_2 and f_3 this FD is not a tree but a DAG with particular characteristics:

- A FD written in OFD is essentially a DAG, that is, a set of *nodes* (let us call it N) and a *relation* between nodes. Nodes are related by decomposition edges ($DE \subseteq N \times N$). Features naturally map to elements of N, and decomposition edges to elements of DE.

- In OFD, there is always a *unique root feature*. We will call it r and thus have $r \in N$. In Figure 8.1, the root is f_1. As the feature f_1 is decomposed into features f_2 and f_3, there we have (f_1, f_2) $\in DE$ and (f_1, f_3) $\in DE$.

- Although decomposition edges are represented in the concrete syntax by plain lines instead of arrows, they do have a direction from top to bottom. In OFD (as well as in most FD languages), the direction is represented by the fact that a feature is placed graphically above its sub-features. This relation is thus *directed*. For instance, the directed edge (f_1, f_3).

- Also, in OFD, the decomposition graph must *have no cycle* (that is less restrictive than having to be a tree, like in OFT).

- Hence, *DE* must form a single-rooted *DAG*.

To abstract the various kinds of decompositions, we introduce a *labelling function* λ that maps each node to a Boolean operator either *and*$_s$ or *xor*$_s$, respectively for *and*- and *xor*-decompositions. s denotes the arity of the operator and must be equal to the number of sub-nodes (sub-features) of the labelled node. The signature of λ is thus $\lambda : N \rightarrow NT$, and NT (node type) is a set of Boolean operators. It contains operators *and*$_1$, *and*$_2$, *and*$_3$, ... as well as operators *xor*$_2$, *xor*$_3$, ...[*] So, if f_1 is *and*-decomposed into f_2 and f_3, we will have $\lambda(f_1)$ = *and*$_2$. The node f_1 is thus called an *and*$_2$-node, or simply *and*-node. The node f_3 is thus called a *xor*$_3$-node, or simply *xor*-node. The arity of $\lambda(f_1)$ is two because f_1 has two sons (f_2, f_3) and the arity of $\lambda(f_3)$ is three because f_3 has three sons (f_5, f_6 and f_7). We adopt similar terminological conventions for nodes labelled with other (types of) operators. Another convention is that *terminal features*, i.e. features that have no sub-feature, are λ-labelled with *and*$_0$. In Figure 8.1, the terminal features are f_4, f_5, f_6, f_8, f_9. Their *and*$_0$ λ-label is not indicated in the abstract syntax to avoid overloading of the figure.

The abstract syntax for *optional features* (those with a hollow circle on top in the concrete syntax) is a little trickier. In Figure 8.1, the main difference between the concrete and abstract syntax is manifest: the abstract syntax possesses one node (f_9?) for the circle. It corresponds to the decomposition of the feature f_9. Indeed, for each optional feature in the concrete syntax (for instance f_9), our abstract syntax possesses two nodes (f_9 and f_9?). f_9? corresponds to the hollow circle and is introduced as an intermediate node between f_9 and its parents. The only son of f_9? is f_9 and $\lambda(f_9?)$ = *opt*$_1$. This definition for optional nodes came after noticing that they actually play a role similar to the *and*- and *xor*-decomposition, except for the fact that this kind of operator only acts

---

[*]In addition, *and*$_1$ = *xor*$_1$ and NT also contains *and*$_0$, *xor*$_0$ and *opt*$_1$.

upon one sub-node.

The last part of OFD's concrete syntax that $\mathcal{L}_{OFD}$ should account for is the textual constraint language, called "Composition Rules" (*CR*) in (Kang et al., 1990). In the concrete diagram, the language is used to specify a (possibly empty) set of rules, located at the bottom of the diagram. In the abstract syntax, we call the set of rules Φ and define it as a set of words obeying the following production rule: *CR* ::= f_1(`requires` | `excludes`)f_2, where f_1, $f_2 \in N$. In Figure 8.1, $\Phi = \{f_4$ `requires` $f_9\}$.

Important concepts we introduced in $\mathcal{L}_{OFD}$ are the concepts of *primitive feature* and *non-primitive feature*. A *primitive feature* is a feature that has an interest per se and that will influence the final products. We do not confuse primitive and terminal features. A *terminal feature* (or *leaf feature*) is a feature without any sub-feature. Following (Batory, 2005), we distinguish *terminal* and *compound* (or non-terminal) features. A *compound feature* is an intermediate node used for decomposition. Terminal features are usually primitive, but compound features can also be designated as primitive too. In the literature, there is currently no agreement on the following questions:

- Are constraints only expressible between primitive features?

- Are all the features in a FD relevant to distinguish two products, or is there a subset of the features that is relevant and one that is not?

- Is this relevant subset limited to terminal features or can it also include non-terminal features?

Actually, these questions primarily address semantics, but have consequences on the syntax. For example, in Figure 6.2, one could question whether the (absence or presence of the) feature `Measures` is useful to describe a product, or if (the absence or presence of) its sub-features, `L/Km` and `Gallon/mile`, suffice. Since we found no agreement on these questions in the literature, we adopted a neutral formalisation. Our solution accounts for the fact that the modeller can consider only part of the features as relevant. Although there is no construct in the concrete syntax (neither for OFD, nor any other FD language we know of), we need to introduce one in the abstract syntax, namely a subset P of N ($P \subseteq N$). We will see the impact of P when we address the semantics of OFD. Finally, although we leave it to the modeller to determine P, we reasonably expect that P contains terminal nodes and does not contain *opt*-nodes. But we do not need to impose these rules.

In the previous paragraphs, we have informally described and justified the constructs of $\mathcal{L}_{OFD}$. Now it is time to define it formally.

Definition 8.1.1 (Original Feature Diagram) *An* OFD $d \in \mathcal{L}_{OFD}$ *is a tuple* $(N, P, r, \lambda, DE, \Phi)$ *where:*

- *N is its set of* nodes*;*

- *$P \subseteq N$ is its set of* primitive nodes*;*

- *$r \in N$ is the* root*;*

- *$\lambda : N \rightarrow NT$ labels each node with an operator from NT, where $NT = and \cup xor \cup \{opt_1\}$. The semantics of each operator will be provided in the semantic function of OFD (Section 8.1.3);*

- $DE \subseteq N \times N$ *is the set of* decomposition edges*;* $(n, n') \in DE$ *is alternatively noted* $n \to n'$*;*

- $\Phi \subseteq CR$ *are the textual constraints.*

Furthermore, d must satisfy the following well-formedness rules:

1. *Only r has no parent:* $\forall n \in N.(\nexists n' \in N.n' \to n) \Leftrightarrow n = r.$

2. *DE is acyclic:* $\nexists n_1, ..., n_k \in N.n_1 \to \ldots \to n_k \to n_1.$

3. *Node operators are of adequate arities:* $\forall n \in N.\lambda(n) = op_k \wedge k = \sharp\{(n, n')|n \to n'\}.$

4. *Terminal Nodes (leaves) are* and_0 *or* xor_0*-labelled*[†]*:* $\forall n \in N.(\nexists n' \in N.n \to n') \Leftrightarrow \lambda(n) = and_0 \vee \lambda(n) = xor_0.$

Note that the first two well-formedness rules above should be enforced at the level of the concrete syntax (e.g., by a graphical OFD editor), whereas the last two rules should be guaranteed when moving from the concrete to the abstract syntax, and the modeller should not care about them.

8.1.2 The semantic domain of OFD ($\mathcal{S}_{OFD}$)

In the surveyed literature, there seems to be an agreement that FDs are meant to represent *sets of products*, and each *product* is seen as a *combination of features*. These tenets were present from the beginning in OFT (Kang et al., 1990) and were adopted without much controversy in its extensions, including OFD (Kang et al., 1998). In particular, none of the surveyed languages attempted to further define the "content" of a feature beyond its name (viz., the labels appearing in the nodes of the FD), except for some recent work (Benavides et al., 2005a; Classen et al., 2008). We published the first formalisation of a semantic domain specifically devoted to FDs in (Bontemps et al., 2004). In this semantic domain, named *PL*, the atomic building blocks are features (nodes), a bit in the same way that propositions are the atomic building blocks in the semantic domain of propositional logic (see, e.g., (Tarski, 1956)). Definition 8.1.2 presents mathematically the notions of *product* and *product line*. This definition relies on the more general notion of *configuration* given in Definition 8.1.4.

Definition 8.1.2 (OFD Semantic Domain) *The semantic domain of OFD is mathematically defined as the* product lines*. A* product line *(PL) is a set of products, i.e., any element of PL = $\mathcal{PPP}$ where*

- *P is the set of primitive nodes.*

- *$\mathcal{PP}$ is the set of all possible products.*

Figure 8.2 gives an illustration of this. A product, say c, is a combination (i.e. a set) of primitive nodes. In this case, c is the set $\{f_1, f_3, f_4, f_5, f_6, f_7\}$. A product line, e.g. *pl*, is a set of products. Here, *pl* is a set of 3 products: $\{\{f_3, f_6\}, \{f_1, f_2, f_3, f_4, f_5, f_6, f_7\}, \{f_1, f_4, f_5, f_7\}\}$.

[†]The xor_0 node is also called the FALSE node that is only necessary for theorem proving (see Theorem 9.1.9).

$c \in \mathcal{P}\, \mathsf{P}$ $\boxed{f1\ f3\ f4\ f5\ f6\ f7}$

$\mathsf{pl} \in \mathcal{P}\, \mathcal{P}\, \mathsf{P}$

$f3\ f6$

$f1\ f2\ f3\ f4\ f5\ f6\ f7$

$f1\ f4\ f5\ f7$

...
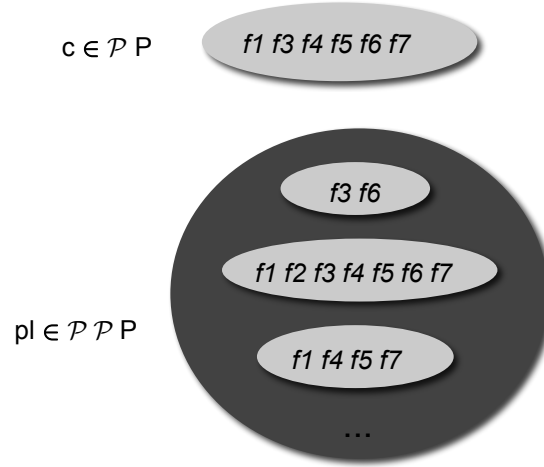
Figure 8.2: OFD's semantic domain: $\mathcal{S}_{OFD}$

However, the definition of semantic domains may differ from *PL*. For example, some authors have proposed to use lists instead of sets (van Deursen and Klint, 2002). The cause usually turns out to be an implementation bias. How to compare *PL* with other semantic domains will be discussed in Chapter 9. For the time being, an important observation is that *PL*, as all semantic domains, is indeed insufficient to describe semantics: it does say *what* is a product line, but it fails to say *which* products pertain to the product line a given OFD stands for. This is the role of the semantic function.

8.1.3 The semantic function of OFD ($\mathcal{M}_{OFD} : \mathcal{L}_{OFD} \to PL$)

In Figure 8.3, we illustrate the semantic function of OFD ($\mathcal{M}_{OFD}$). To every diagram *d*, it assigns a product line, noted $\mathcal{M}_{OFD}[\![d]\!]$, in the semantic domain *PL*. $\mathcal{M}_{OFD}[\![d]\!]$ is formally described in Definitions 8.1.3 and 8.1.5. Definition 8.1.3 indicates *which* set of products is returned by $\mathcal{M}_{OFD}[\![d]\!]$: the set of the configurations (combinations of nodes) that are *valid* wrt *d*, restricted to their primitive nodes. Definition 8.1.5 provides four clear and compact rules telling what in OFD is a valid configuration wrt *d*. The fact that a configuration *c* is valid wrt *d* is noted $c \vDash d$.

Definition 8.1.3 (OFD Semantic function) *The semantics of an OFD d is a product line (Definition 8.1.2) consisting of the products of d, i.e. its valid configurations (Definitions 8.1.4 and 8.1.5) restricted to primitive nodes:* $\mathcal{M}_{OFD}[\![d]\!] = \{c \bigcap P | c \vDash d\}$

Definition 8.1.4 (Configuration) *A* configuration *is a set of nodes, i.e., any element of* $\mathcal{P}N$.

Definition 8.1.5 (Valid configuration) *A configuration (Definition 8.1.4) $c \in \mathcal{P}N$ is* valid *for a $d \in \mathcal{L}_{OFD}$, noted $c \vDash d$, iff:*

   1. The root is in: $r \in c$ and the node operator labelling r is TRUE;
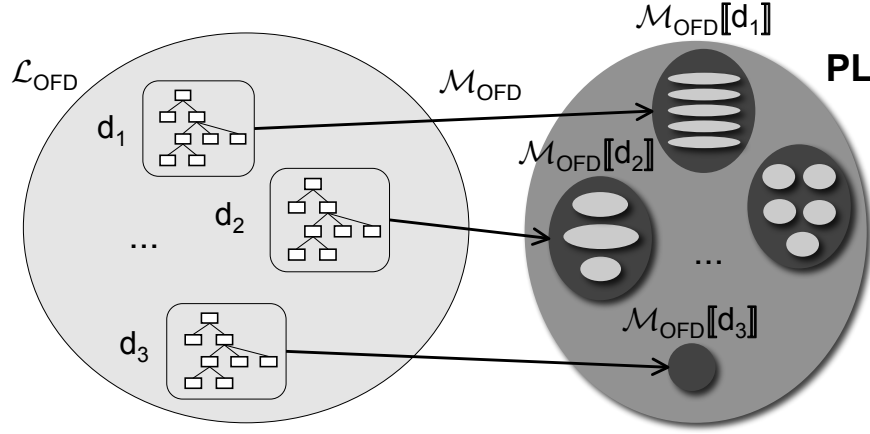
Figure 8.3: OFD's semantic function: $\mathcal{M}_{OFD}$

2. *The meaning of nodes is satisfied: If a node $n \in c$, and n has sons $s_1, \ldots, s_k$ and $\lambda(n) = op_k$, then $op_k(s_1 \in c, \ldots, s_k \in c)$ must evaluate to TRUE. This evaluation depends on the possible values of $NT = and \cup xor \cup \{opt_1\}$ that are mapped to Boolean functions (operators) where:*

   - *and is the set of operators and_s ($s \in \mathbb{N}$), that return TRUE iff all their s arguments are TRUE;*
   - *xor is the set of operators xor_s ($s \in \mathbb{N} \setminus \{0, 1\}$) that return TRUE iff exactly one of their s arguments is TRUE;*
   - *opt_1 is the operator that returns TRUE;*

3. *The configuration must satisfy all textual constraints: $\forall \phi \in \Phi, c \vDash \phi$, where $m \vDash \phi$ means that we replace each node name n in ϕ by the truth value of $n \in c$, evaluate ϕ and get TRUE. Namely:*

   - *if ϕ is a CR constraint of the form f_1 requires f_2, we say that $m \vDash \phi$ iff $(f_1 \in c) \Rightarrow (f_2 \in c)$ evaluates to TRUE;*
   - *if ϕ is a CR constraint of the form f_1 excludes f_2, we say that $m \vDash \phi$ iff $and_2(f_1 \in c, f_2 \in c)$ evaluates to FALSE.*

4. *If f is in the configuration and f is not the root, one of its parents n, called its* justification, *must be too: $\forall f \in N.f \in c \wedge f \neq r: \exists n \in N : n \in c \wedge n \to f$.*

When $\mathcal{M}_{OFD}[\![d]\!]$ returns an empty set of products, i.e. the empty *PL*, it means that d is *non-satisfiable* (or, *inconsistent*). In Figure 8.3, this is the case for d_3. This happens when there is no

product combination that can satisfy the constraints in d. Checking consistency, as well as many other tasks, can usually not be performed efficiently just by processing syntax, nor by letting the modeller inspect the diagram. Hence, our semantics enables a correct and faithful implementation of time-consuming and error-prone tasks.

In Figure 8.4, we now take a closer look at the product validity rules of Definition 8.1.5.

- The left part of the figure presents the abstract syntax d_1 of the FD, written in OFD, that is identical to the right part of Figure 8.1. We assume that all nodes in the abstract syntax represented by d_1 are primitive nodes, except for "f_9?" that is an optional node generated to account for an optional feature in the concrete syntax.

- The right part of the figure presents the semantics of d_1 ($\mathcal{M}_{OFD}[\![d_1]\!]$) in the semantic domain PL. The application of the semantic function ($\mathcal{M}_{OFD}$) on d_1 results in a product line composed of two products: $\{\{f_1, f_2, f_3, f_4, f_7, f_8, f_9\}, \{f_1, f_2, f_3, f_5\}\}$.
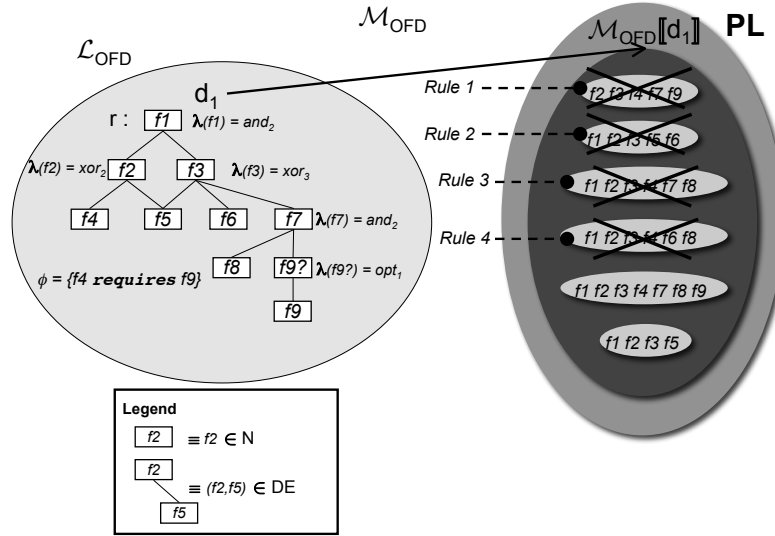
Figure 8.4: OFD's semantics: validity rules

Furthermore, we illustrate in Figure 8.4 why some products are discarded according to the four rules presented in Definition 8.1.5:

1. The first rule (Rule 1) imposes that the root ($r = f_1$) appears in every valid product. Hence, the product $\{f_2, f_3, f_4, f_7, f_8, f_9\}$, for instance, cannot be part of the product line ($\{f_2, f_3, f_4, f_7, f_8, f_9\} \notin \mathcal{M}_{OFD}[\![d_1]\!]$).

2. The second rule (Rule 2) describes the semantics of the Boolean operators coming from the decomposition edges and from the optional features. This rule relies on the semantics of the Boolean operator opt_1 as well as the operators in *and* and *xor*. Their semantics has been

recalled in point 2 of Definition 8.1.5. In Figure 8.4, the second rule is applied to discard the product $\{f_1, f_2, f_3, f_5, f_6\}$, say c, because f_3 appears in it together with more than one node among f_5, f_6 and f_7. Indeed, f_3 is labelled with xor_3 and has 3 sons: f_5, f_6 and f_7. In c, $xor_3(f_5 \in c, f_6 \in c, f_7 \in c)$ would then evaluate to FALSE.

3. The third rule (Rule 3) is similar in spirit to the former, except that it deals with the operators (`requires` and `excludes`) appearing in Φ, the *CR* textual constraints accompanying the graphical part of the OFD. When applied to the example, the rule interprets the CR f_4 `requires` f_9 by checking the truth value of $(f_4 \in c) \Rightarrow (f_9 \in c)$, which in the case of the product $c = \{f_1, f_2, f_3, f_4, f_7, f_8\}$ evaluates to FALSE.

4. The fourth and last rule (Rule 4) is called the *justification* rule. It guarantees that, except for the root, a node cannot be present in a valid product without at least one of its parent nodes being present as well. It says "at least one of its parents" because OFDs are DAGs and a node can therefore have multiple parents. In the example, this rule discards the product $\{f_1, f_2, f_3, f_4, f_6, f_8\}$ because f_8 belongs to c but f_7, its only parent, does not. The justification rule has often been overlooked in the literature. This leads to strongly counter-intuitive semantics. Without the justification rule, the OFD in Figure 8.4 would accept, e.g., products $\{f_1, f_2, f_3, f_5, f_8, f_9\}$ or $\{f_1, f_2, f_3, f_4, f_6, f_8\}$ as part of its semantics. Justifications also explain the difference between an *and*-decomposition of a feature into sub-features and a `requires` constraint between two features: the presence of sub-feature is justified by its parent, while a `requires` gives no justification.

When all the rules in Definition 8.1.5 are satisfied and when all the non-primitive nodes in the products have been discarded according to Definition 8.1.3, we see that the semantics of the OFD in Figure 8.4 comes unambiguously as the following product line, made of two valid products: $\{\{f_1, f_2, f_3, f_4, f_7, f_8, f_9\}, \{f_1, f_2, f_3, f_5\}\}$.

8.2 Formal Configurable Definition of FD Languages

Once OFD has been formally defined, its definition can be used as a starting point to define a family of FD languages, called FFD. The idea is to extend the OFD definition in order to integrate other FD languages. Obviously this extended definition should still follow Harel and Rumpe's principles (Chapter 4) but however they need to be slightly adapted. Hence, we first show in Section 8.2.1 how we adapt these principles to define FFD and, then, we apply these principles in Section 8.2.2 to provide a formal and configurable definition to FFD.

8.2.1 How to define Free Feature Diagram (FFD)

The definition of FFD is no exception and should still follow Harel and Rumpe's principles (Chapter 4). However, these principles have been proposed to formally define one language while our purpose is to define a family of languages. Consequently, some adaptations are necessary:

1. Firstly, the scope of the family should be defined to determine which FD languages are members of FFD. Initially, the scope of FFD contains our formal language VFD (Bontemps et al.,

2004; Schobbens et al., 2007) and the informal FD languages gathered in the state of the art (Chapter 6): OFT (Kang et al., 1990), OFD (Kang et al., 1998), RFD (Griss et al., 1998), GPFT (Eisenecker and Czarnecki, 2000), EFD (Riebisch et al., 2002), VBFD (van Gurp et al., 2001), PFT (Eriksson et al., 2005).

2. Secondly, we need to adapt the principles proposed by Harel and Rumpe (Chapter 4). These adaptations are illustrated in Figure 8.5 and should be analysed in comparison with the original concepts already described in Figure 4.1. The main adaptation concerns the syntactic domain that should be generic to take into account all the members contained in the scope of our family of languages. The left part of the figure illustrates this *generic syntactic domain* ($\mathcal{L}_{FFD}$, see Definition 8.2.1) that enables to describe, not one single syntactic domain for one language, but syntactic domains of all the members of the FFD family ($\mathcal{L}_{OFT}, \mathcal{L}_{OFD}, \ldots, \mathcal{L}_{PFT} \in \mathcal{L}_{FFD}$).
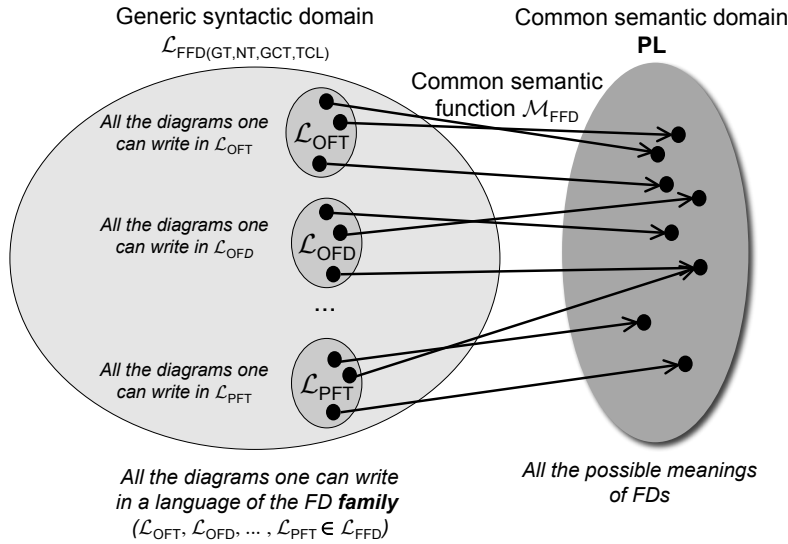
Figure 8.5: Meaningful Modelling for FD Family

Once this generic syntactic domain is defined, the semantic domain and function are now common for OFT (Kang et al., 1990) and most of its extensions (Kang et al., 1998; Griss et al., 1998; Eisenecker and Czarnecki, 2000; Riebisch et al., 2002; van Gurp et al., 2001; Eriksson et al., 2005; Bontemps et al., 2004). Hence, this new approach allows managing a family of languages reusing the semantics that we have already defined for OFD (Section 8.1).

8.2.2 Free Feature Diagram (FFD) Definition

In this section, we formally define FFD and more specifically its generic syntactic domain (Section 8.2.2.1), common semantic domain (Section 8.2.2.2) and common semantic function (Section 8.2.2.3).

8.2.2.1   FFD Syntactic Domain ($\mathcal{L}_{FFD}$)

The definition of the generic *syntactic domain* ($\mathcal{L}_{FFD}$) is a generalisation of OFD *syntactic domain* (Section 8.1). It should be adapted to cover not only OFD but all the FD languages included in FFD. The resulting generic *syntactic domain* (Definition 8.2.1) is defined with a parametric construction that generalises the syntax of all FD variants. The FFD abstract syntax definition is parametrised according to four variation points revealed by the comparison of OFD with the other FD languages covered by FFD. Accordingly, the FFD abstract syntax is parameterized and is of the form: $FFD(GT, NT, GCT, TCL)$ where:

- *Parameter GT*. The decomposition edges in a FD can form a tree or a DAG. For this, we introduce the parameter GT: GT (Graph Type) is either DAG or TREE. In OFD, the decomposition edges form a DAG. For the studied FD languages, the values for this parameter are given in Figure 8.6.
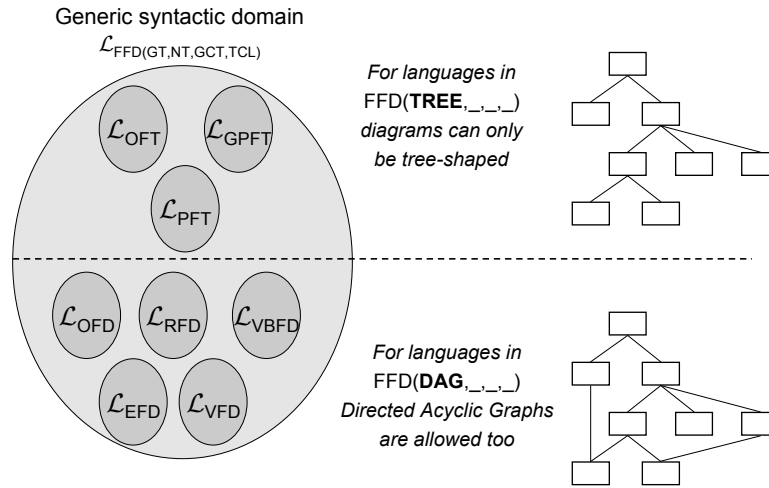
Figure 8.6: Variation Point: Graph Type (GT)

- *Parameter NT*. The type of Nodes in a FD may vary. The different types of nodes found in the literature are: *and*, *xor*, opt_1, *or* and *card*.

  Example 8.2.1 *In the running example used in Chapter 6, we have already encountered different types of nodes (see Figure 8.7):*

  - *The node* `Monitor Engine Performance` *in Figure 8.7(a) is a node of type and, i.e. a and-node. Since it has 3 sons, we will use the operator of this arity, and_3. We say that it bears the operator and_3.*
  - *The node* `Methods` *in Figure 8.7(b) bears operator or_3.*
  - *The node* `Measures` *in Figure 8.7(c) bears operator xor_2.*

– *The nodes* `Monitor Engine Performance`, `Measures` *and* `Methods` *in Figure 8.7 also bear, respectively, operators* $card_3[3..3]$, $card_2[1..1]$ *and* $card_3[1..3]$.
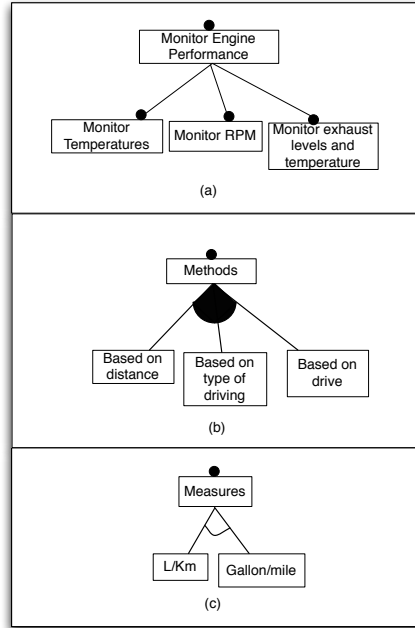
Figure 8.7: Examples of and_3, or_3, xor_2 in GPFT

For this, we introduce the parameter NT: NT (Node Type) is a set of Boolean functions (operators): *and*, *xor*, *or*, *opt* and *card*. In OFD, the type of nodes allowed are *xor*-node, *and*-node and opt_1-node. *Or*-node and *card*-node are also included in some other languages whereas *xor*-node is not available in others. It is possible to add more node types by defining Boolean operators. For the studied FD languages, the values for this parameter are given in Figure 8.8.

• *Parameter GCT*. The type of graphical constraints may vary. Some languages do not have any graphical constraints whereas others have *requires* ($\Rightarrow$) and *excludes* (($|$)) graphical constraints. For this, we introduce the parameter GCT: GCT (Graphical Constraint Type) is a set of binary Boolean operators. E.g.: *Requires* ($\Rightarrow$) or *Mutex* ($|$). In OFD, no graphical constraints are allowed. Other graphical constraints between features that are not boolean constraints exist such as generalisation or implementation (Kang et al., 1998). However, since we are not able to give them a semantics we prefer, at this point, not to consider them as possible values for GCT. For the studied FD languages, the values for this parameter are given in Figure 8.9.

• *Parameter TCL*. The textual constraint language may vary. Some languages do not have any textual constraints whereas others (re)use constraint languages. For this, we introduce the
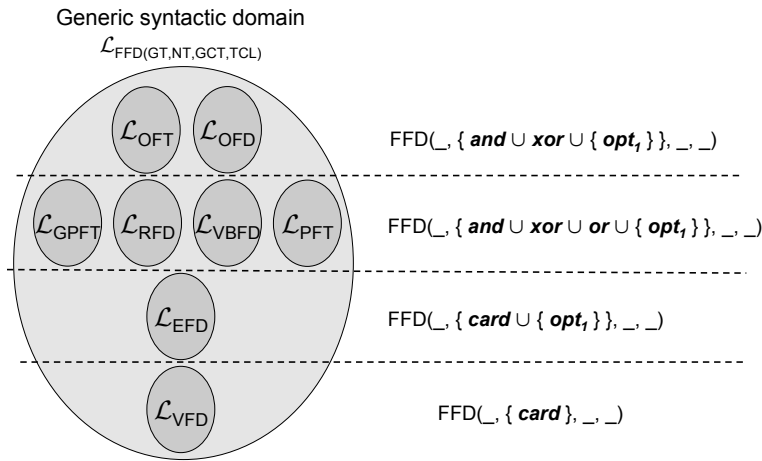
Figure 8.8: Variation Point: Node Type (NT)
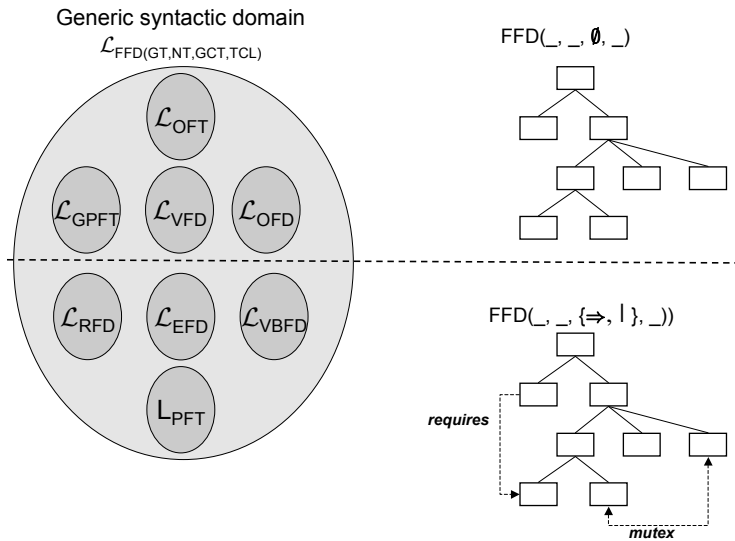
Figure 8.9: Variation Point: Graphical Constraint Type (GCT)

parameter TCL: TCL (Textual Constraint Language) is a subset of the language of Boolean formulae where the atomic predicates are the nodes of the FD. The main difference with the previous parameter is that the semantics of the constraint languages are defined independently from the FD language itself while the semantics of the graphical constraints still need to be explicitly defined within the FD language. In OFD, the textual constraint language is "Composition Rules (CR)" (Kang et al., 1990, p.71), $CR ::= f_1(requires \mid excludes)f_2$ where

f_1, f_2 boolean variables correspond to features (Section 8.1.1). In addition, more powerful languages, such as propositional logic, can be included in the FD language (Batory, 2005). For the studied FD languages, the values for this parameter are given in Figure 8.10.

Generic syntactic domain
$\mathcal{L}_{FFD(GT,NT,GCT,TCL)}$

$\mathcal{L}_{OFT}$

$\mathcal{L}_{GPFT}$ $\mathcal{L}_{EFD}$ $\mathcal{L}_{OFD}$

$\mathcal{L}_{RFD}$ $\mathcal{L}_{VBFD}$

$\mathcal{L}_{VFD}$ $\mathcal{L}_{PFT}$

FFD(_, _, _, **CR**)

where
CR is the language of *composition rules*

CR ::= p1 (**requires** | **mutex**) p2

FFD(_, _, _, $\emptyset$)
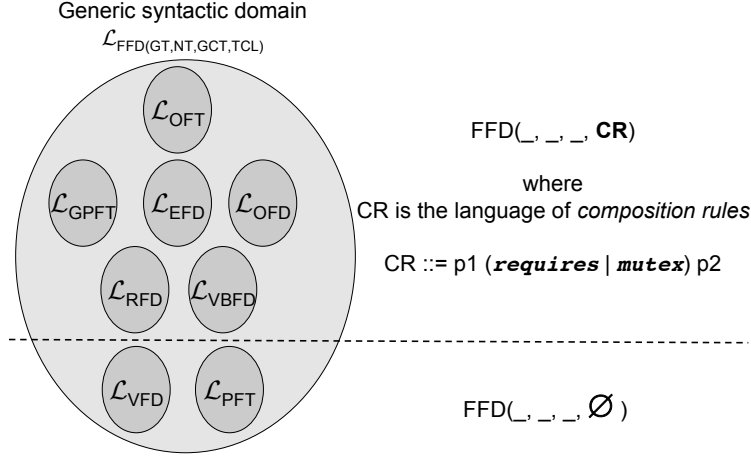
Figure 8.10: Variation Point: Textual Constraint Language (TCL)

The values of these four parameters are summarised in Table 8.1 for the studied FD languages. Other combinations of values exist and determine other possible FD languages. Defining a language now boils down to filling in a row of Table 8.1. Similarities and differences in the abstract syntax of the languages can also be studied. A prominent example is that the FD languages RFD (Griss et al., 1998) and VBFD (van Gurp et al., 2001)) appear to be identical in our definition (Line 3 in Table 8.1).

Short Name	References	GT	NT	GCT	TCL
OFT	Kang et al. (1990)	TREE	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
OFD	Kang et al. (1998)	DAG	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
RFD=VBFD	Griss et al. (1998); van Gurp et al. (2001)	DAG	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, |\}$	CR
EFD	Riebisch et al. (2002); Riebisch (2003)	DAG	$card \cup \{opt_1\}$	$\{\Rightarrow, |\}$	CR
GPFT	Eisenecker and Czarnecki (2000)	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\emptyset$	CR
PFT	Eriksson et al. (2005)	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, |\}$	$\emptyset$
VFD	Bontemps et al. (2004); Schobbens et al. (2007)	DAG	$card$	$\emptyset$	$\emptyset$

Table 8.1: Family of Existing FD languages

When comparing definitions of FFD syntactic domain ($\mathcal{L}_{FFD}$) (Definition 8.2.1) and OFD syntactic domain ($\mathcal{L}_{OFD}$) (Definition 8.1.1), we can underline three main differences:

- $\mathcal{L}_{FFD}$ is defined according to four parameters (GT, NT, GCT, TCL) and their associated values described previously.

- Two other types of nodes are allowed in $\mathcal{L}_{FFD}$: *or* and *card*.

- $\mathcal{L}_{FFD}$ allows graphical constraints formalised as Constraint Edges (CE).

Definition 8.2.1 (Free Feature Diagram (FFD)) *A FFD $d \in FFD(GT, NT, GCT, TCL) = (N, P, r, \lambda, DE, CE, \Phi)$ where:*

- *GT (Graph Type) is either* DAG *or* TREE*;*

- *NT (Node Type) is a set of node types where $NT = and \cup xor \cup \{opt_1\} \cup or \cup card$. Each node type is mapped to a set of commutative Boolean operators, at most one per arity;*

- *GCT (Graphical Constraint Type) is a set of binary Boolean operators. E.g.:* Requires *($\Rightarrow$) or* Mutex *(|);*

- *TCL (Textual Constraint Language) is a subset of the language of Boolean formulae where the atomic predicates are the nodes of the FD;*

- *N is its set of* nodes*;*

- *$P \subseteq N$ is its set of* primitive nodes*;*

- *$r \in N$ is the* root *of the FD, also called the* concept*;*

- *$\lambda : N \rightarrow NT$ labels each node with an operator from NT;*

- *$DE \subseteq N \times N$ is the set of* decomposition edges*; $(n, n') \in DE$ will rather be noted $n \rightarrow n'$;*

- *$CE \subseteq N \times GCT \times N$ is the set of* constraint edges*;*

- *$\Phi \subseteq TCL$ are the textual constraints.*

In addition, a FD must obey the following constraints:

1. Only r has no parent: $\forall n \in N.(\nexists n' \in N.n' \rightarrow n) \Leftrightarrow n = r$.

2. DE is acyclic: $\nexists n_1, ..., n_k \in N.n_1 \rightarrow ... \rightarrow n_k \rightarrow n_1$.

3. If $GT =$ TREE, DE is a tree: $\nexists n_1, n_2, n_3 \in N.n_1 \rightarrow n_2 \wedge n_3 \rightarrow n_2 \wedge n_1 \neq n_3$.

4. Node operators are of adequate arities: $\forall n \in N.\lambda(n) = op_k \wedge k = \sharp\{(n, n')|n \rightarrow n'\}$.

5. Terminal Nodes are and_0 or xor_0-labelled[‡]: $\forall n \in N.(\nexists n' \in N.n \rightarrow n') \Leftrightarrow \lambda(n) = and_0 \vee \lambda(n) = xor_0$.

[‡]The xor_0 node is also called the FALSE node and is only necessary for theorem proving (see Theorem 9.1.9).

8.2.2.2 FFD Semantic Domain ($\mathcal{S}_{FFD}$)

The *Semantic Domain* of FFD ($\mathcal{S}_{FFD}$, Definition 8.2.2) is identical to the one defined for OFD ($\mathcal{S}_{OFD}$, Definition 8.1.2).

Definition 8.2.2 (FFD Semantic Domain) *The semantic domain of FFD is mathematically defined as the* product lines. *A product line (PL) is a set of products, i.e., any element of PL = $\mathcal{PPP}$ where*

- *P is the set of primitive nodes.*

- *$\mathcal{PP}$ is the set of all possible products.*

8.2.2.3 FFD Semantic function ($\mathcal{M}_{FFD} : \mathcal{L}_{FFD} \rightarrow \mathcal{PPP}$)

In Figure 8.5, we illustrate $\mathcal{M}_{FFD}$, the semantic function of FFD. To every diagram d, written in any FD language contained in the language family, it assigns a *PL* noted $\mathcal{M}_{FFD}[\![d]\!]$. $\mathcal{M}_{FFD}[\![d]\!]$ is formally described in Definitions 8.2.3 and 8.2.5. Definition 8.2.3 indicates *which* set of products is returned by $\mathcal{M}_{FFD}[\![d]\!]$: the set of the configurations (combinations of features) that are *valid* wrt d, restricted to their primitive features. Definition 8.2.5 provides five clear and compact rules telling what in FFD is a valid configuration wrt d. The fact that a configuration c is valid wrt d is noted $c \vDash d$.

Definition 8.2.3 (FFD Semantic function) *The semantics of an FFD d is a product line (Definition 8.2.2) consisting of the products of d, i.e. its valid configurations (Definitions 8.2.4 and 8.2.5) restricted to primitive features/nodes:* $\mathcal{M}_{FFD}[\![d]\!] = [\![d]\!] = \{c \cap P | c \vDash d\}$

Definition 8.2.4 (Configuration) *A* configuration *is a set of nodes, i.e., any element of $\mathcal{PN}$.*

Definition 8.2.5 (Valid configuration) *A configuration $c \in \mathcal{PN}$ is* valid *for a $d \in \mathcal{L}_{FFD}$, noted $c \vDash d$, iff:*

1. *The root is in: $r \in c$ and the node operator labelling r is TRUE;*

2. *The meaning of nodes is satisfied: If a node $n \in c$, and n has sons $s_1, \ldots, s_k$ and $\lambda(n) = op_k$, then $op_k(s_1 \in c, \ldots, s_k \in c)$ must evaluate to TRUE.*

   - and *maps to the set of operators* and$_s$ *that return TRUE iff all their s arguments are TRUE;*

   - xor *maps to the set of operators* xor$_s$ *that return TRUE iff exactly one of their s arguments is TRUE;*

   - opt$_1$ *is the only operator in* opt. *It always returns TRUE. By providing only one operator, we restrict these nodes to have only one son.*

   - or *maps to the set of operators* or$_s$ *that return TRUE iff some of their s arguments are TRUE;*

   - card[i..j] *maps to the set of operators* card$_s$[i..j] *where* $i \in \mathbb{N}$ *and* $j \in \mathbb{N} \cup \{*\}$ *that return TRUE iff at least* i *and at most* j *of their arguments are TRUE.*

3. *The configuration must satisfy all textual constraints: $\forall \phi \in \Phi, c \vDash \phi$, where $m \vDash \phi$ means that we replace each node name n in ϕ by the truth value of $n \in c$, evaluate ϕ and get* TRUE. *Namely:*

   - *if ϕ is a CR constraint of the form f_1* `requires` *f_2, we say that $m \vDash \phi$ iff $(f_1 \in c) \Rightarrow (f_2 \in c)$ evaluates to* TRUE;

   - *if ϕ is a CR constraint of the form f_1* `excludes` *f_2, we say that $m \vDash \phi$ iff $and_2(f_1 \in c, f_2 \in c)$ evaluates to* FALSE.

4. *The configuration must satisfy all graphical constraints: $\forall(n_1, op_2, n_2) \in CE$, $op_2(n_1 \in m, n_2 \in m)$ must be* TRUE.

5. *If s is in the configuration and s is not the root, one of its parents n, called its* justification, *must be too: $\forall s \in N.s \in c \wedge s \neq r: \exists n \in N : n \in c \wedge n \rightarrow s$.*

When comparing definitions of FFD semantic function ($\mathcal{M}_{FFD}$) (Definition 8.2.3) and OFD semantic function ($\mathcal{M}_{OFD}$) (Definition 8.1.3), we can underline two main differences:

- Two other types of nodes are allowed in $\mathcal{L}_{FFD}$: *or* and *card*. Therefore, their respective semantics is defined in rule 2 of Definition 8.2.5.

- $\mathcal{L}_{FFD}$ allows graphical constraints represented as Constraint Edges (CE). Therefore, the semantics of graphical constraints is defined in the new rule 4 of Definition 8.2.5.

8.3 Instantiating FFD

Each surveyed FD language can now be defined by providing adequate values to the parameters of FFD (Table 8.1). Semantics, on the other hand, is given once for all the members of FFD (see Section 8.2.2.3). The formal semantics of a particular FD language defined through FFD thus comes for free. The transformation from one FD language to the FFD structure or *syntactic domain* is a translation from concrete to abstract syntax. For instance, Figure 8.11 illustrates the translation from OFD concrete syntax to FFD abstract syntax: $\mathcal{L}_{OFD} = \mathcal{L}_{FFD}(DAG, \{and \cup xor \cup \{opt_1\}\}, \{\}, CR)$. Indeed, OFD allows DAGs with optional and mandatory features. These features could be decomposed with an *and*-decomposition or a *xor*-decomposition. In addition, OFD allows *requires* and *excludes* textual constraints between features. Another example is provided in Figure 8.12 that illustrates the translation from EFD concrete syntax to FFD abstract syntax: $\mathcal{L}_{EFD} = \mathcal{L}_{FFD}(DAG, \{card \cup \{opt_1\}\}, \{\rightarrow, |\}, CR)$.

The values of the parameters used to instantiate FFD into OFD and EFD indicates how both languages differ. The first difference is the presence of graphical constraints ($\rightarrow$ for *requires*, | for *excludes*) in EFD. The second difference is the substitution in *NT* of the *and* and *xor* operators by the *card* operator. These differences concern the abstract syntaxes of the languages and reveal how they can be distinguish without considering aesthetic differences (concrete syntax). In our research context, examining how a *xor*-decomposition is represented graphically (see Figure 8.13) is not relevant. Nevertheless, examining whether the *xor*-decomposition is allowed or whether another
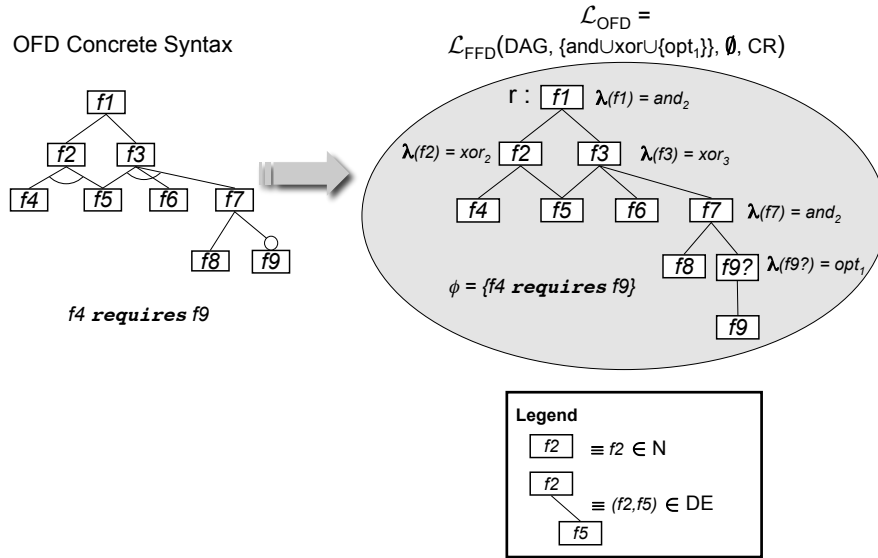
Figure 8.11: From OFD to FFD
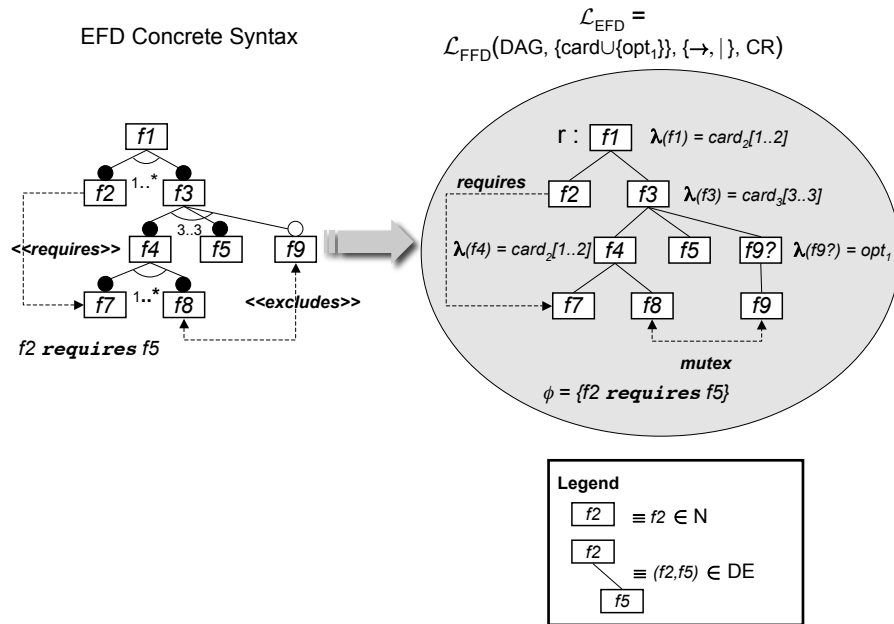
Figure 8.12: From EFD to FFD

construct stands for it is capital. Finally, we should notice that a pair of FDs, one written in OFD and the other in EFD, may share the same translation to FFD. Therefore, they also share the same semantics. However, two FDs that do not map to the same FFD can also share the same semantics. In both cases, we say that they are semantically equivalent.
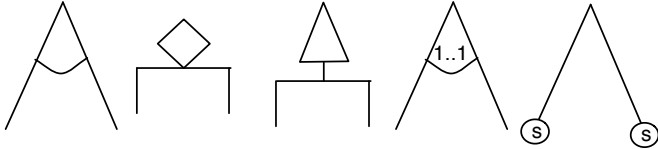
Figure 8.13: Xor-decomposition from Abstract to Concrete Syntax

From Table 8.1, we can immediately observe that some FD languages are syntactically richer than others. For instance, $\mathcal{L}_{OFD} \subset \mathcal{L}_{RFD}$. Indeed, both languages (1) allow DAG and (2) share the same constraint language. However, $\mathcal{L}_{RFD}$ admits the *or* operator and graphical constraints whereas $\mathcal{L}_{OFD}$ does not. The results concerning syntactic inclusion of FD languages are presented in Theorem 8.3.1 and illustrated in Figure 8.14.

Theorem 8.3.1 *We observe the following syntactical inclusions:*

1. *$\mathcal{L}_{OFT} \subset \mathcal{L}_{OFD} \subset \mathcal{L}_{RFD} \subset \mathcal{L}_{FFD}$*

2. *$\mathcal{L}_{OFT} \subset \mathcal{L}_{GPFT} \subset \mathcal{L}_{RFD} \subset \mathcal{L}_{FFD}$*

3. *$\mathcal{L}_{PFT} \subset \mathcal{L}_{RFD} \subset \mathcal{L}_{FFD}$*

4. *$\mathcal{L}_{VFD} \subset \mathcal{L}_{EFD} \subset \mathcal{L}_{FFD}$*

Nevertheless, Theorem 8.3.1 should not be over-interpreted. From the syntactic point of view, we can observe that $\mathcal{L}_{RFD}$ includes most of the other FD languages except $\mathcal{L}_{EFD}$ and $\mathcal{L}_{VFD}$. We also notice that $\mathcal{L}_{VFD}$ and $\mathcal{L}_{OFT}$ seem to be minimal but that no intersection exists between them. The quality of a language does not only depend on the constructs it allows or not. The syntactic analysis is not a sufficient criteria to evaluate languages. Once a language is syntactically included in another language, it does say anything about their respective quality. In addition, when two languages have no syntactic intersection, no conclusion can be drawn. A language with a large number of constructs is not necessarily of better quality than a language with a minimal number of constructs and conversely. Some constructs may not be used or even two constructs may be redundant.
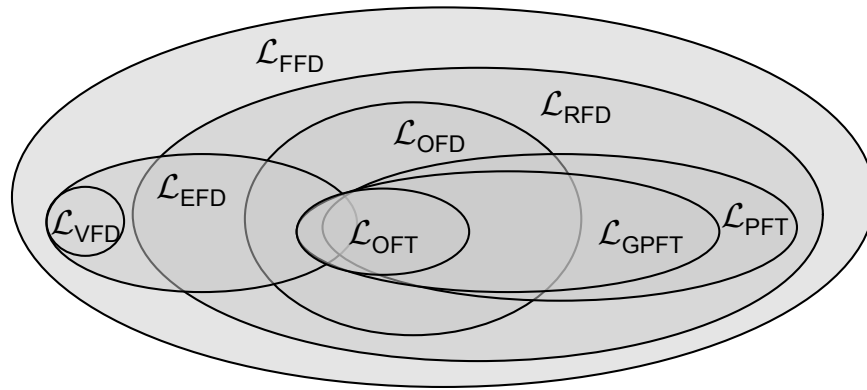
Figure 8.14: FD languages: Syntactic Inclusion

At this stage, the reader should keep in mind that FFD is not a user notation, that is, a notation meant to be used by analysts to draw FD. FFD aims at including the studied FD languages with their respective constructs. By this mean, FFD provides a coherent and unified semantics to all of included constructs but does not question their relevance. Based on this semantics, the languages included in FFD will now become comparable according to the various criteria defined in Chapter 5. Hence, FFD is a formal framework to be applied by method engineers and scientists to formally define, study and compare FD languages. FFD is intended to facilitate these tasks and to increase their rigour and accuracy.

8.4 Semantic issues

Defining a formal semantics as we just did for FFD imposes to take decisions and gives the opportunity to uncover some important issues that might well have remained unnoticed. If such issues are not made explicit and solved before a language is made public (resp. building tools), interpretations of the user (resp. developer) might differ from the intended one, possibly leading to misunderstandings and faulty developments. It is important to note that we do not claim to have provided the *right* semantics for all these languages. We have formalised one semantics that should now be discussed according to these issues and the semantic choices we have made.

8.4.1 Node or Edge-based Semantics

The semantic definition of every language composed of edges and nodes could be based on edge or node transformations. In most studied FD languages, the semantics seems to be node-based with mandatory and optional nodes. However, the semantics of EFD (Riebisch et al., 2002) is edge-based with mandatory and optional edges.

At the *syntactic level*, both solutions are clearly relevant and therefore, for generality, we propose specific decompositions for optional nodes as presented in Figure 8.15. Consider Figure 8.15

(a), a very basic FD. OFD (Kang et al., 1998) seems to hint that it should be abstracted to (b), while EFD (Riebisch et al., 2002) to (c). Because we want to account for both, we take the finer decomposition (d), adding an opt_1-node f_1? under node f_0. The opt_1-node must have f_1 as a son, thus we also add a new decomposition edges in the abstract syntax between f_1? and f_1.
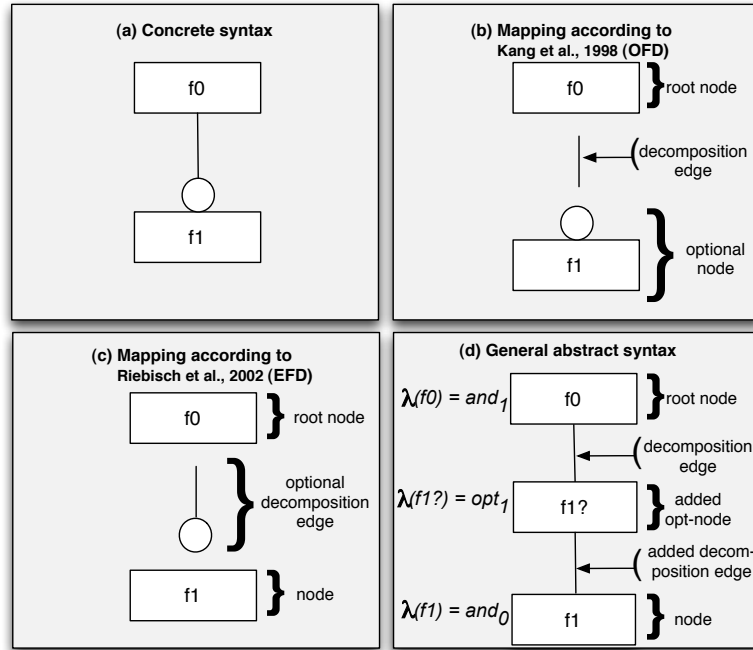
Figure 8.15: Optionality: Three possible abstract syntaxes

At the *semantic level*, the two apparently equivalent FDs in Figure 8.16 actually are not: one is written in OFD with a node-based semantics while the other is written in EFD with an edge-based semantics hinted at (Riebisch et al., 2002). If all features are considered primitive, the OFD allows two products: $\{f_0, f_1, f_2, f_3, f_5\}$ and $\{f_0, f_1, f_2, f_4\}$. Meanwhile, the EFD allows two additional products: $\{f_0, f_1, f_2, f_3, f_4\}$ and $\{f_0, f_1, f_2, f_4, f_5\}$. This difference only occurs when FDs are DAGs. Indeed, the node-based semantics for *xor*-node implies that once the feature f_3 or f_5 has been selected for a product, the feature f_4 will not be reachable. On the contrary, the edge-based semantics for *xor*-decomposition does not prevent that when the edge (f_1, f_3) is selected, the edge (f_2, f_4) is not. Similarly, both edges (f_1, f_4) and (f_2, f_5) may be selected for the same product.

These semantics are clearly different. Our choice in the OFD and FFD definitions is to follow a node-based semantics. However, as illustrated in Figures 8.17 and 8.18, an edge-based semantics can be emulated by a node-based semantics. The idea is first to translate EFD into FFD as previously described (see Figure 8.17) and then to apply a transformation $\mathcal{T}$ on this abstract syntax (see Figure 8.18) before computing the semantics. This transformation $\mathcal{T}$ replaces each node shared by several parents with a non-primitive *and*-node for each incoming edge, each of these *and*-nodes having only one son which is the shared node.

OFD   EFD
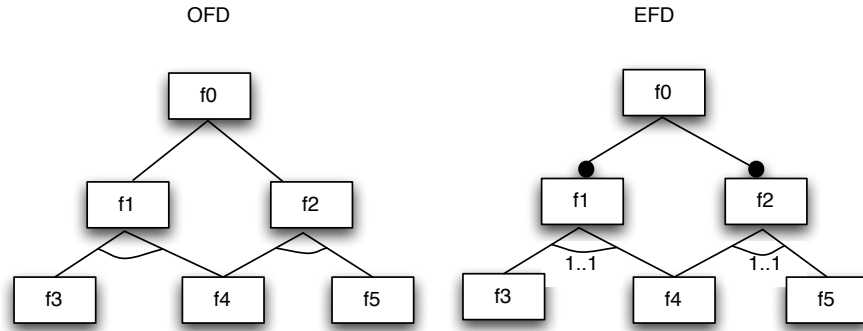
Figure 8.16: Node- *vs.* edge-based semantics
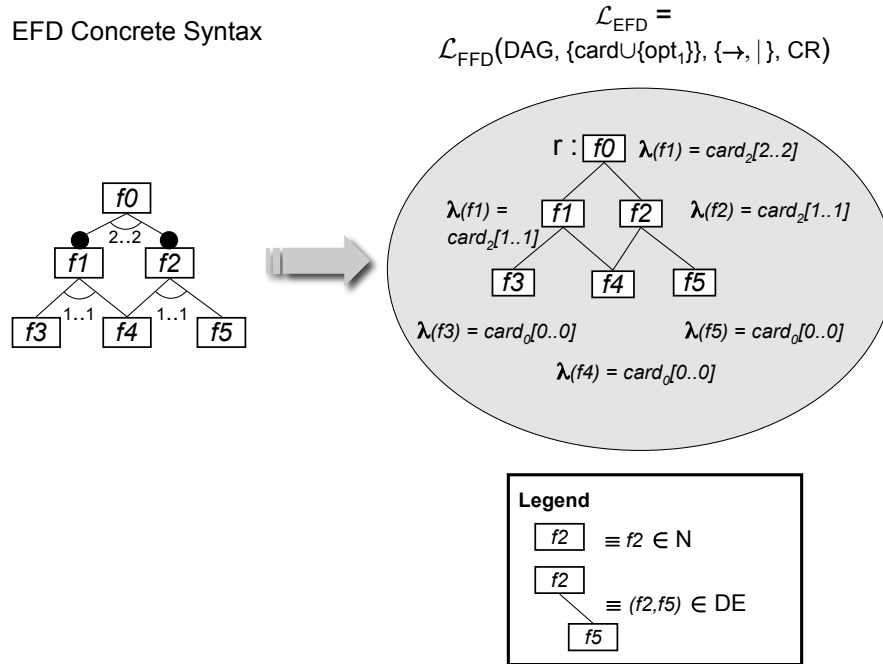
Figure 8.17: From EFD to FFD

For instance, in Figure 8.17 the node f_4 is shared by f_1 and f_2. Therefore two *and*-nodes f_4' and f_4'' ($\lambda(f_4') = card_1[1..1]$ and $\lambda(f_4'') = card_1[1..1]$) sharing son f_4 are added and each incoming edge of f_4 is linked back to them (Figure 8.18). At the end, the node-based semantics corresponding to the translated diagram determines four valid products: $\{f_0, f_1, f_2, f_3, f_5\}$, $\{f_0, f_1, f_2, f_4\}$, $\{f_0, f_1, f_2, f_3, f_4\}$ and $\{f_0, f_1, f_2, f_4, f_5\}$.
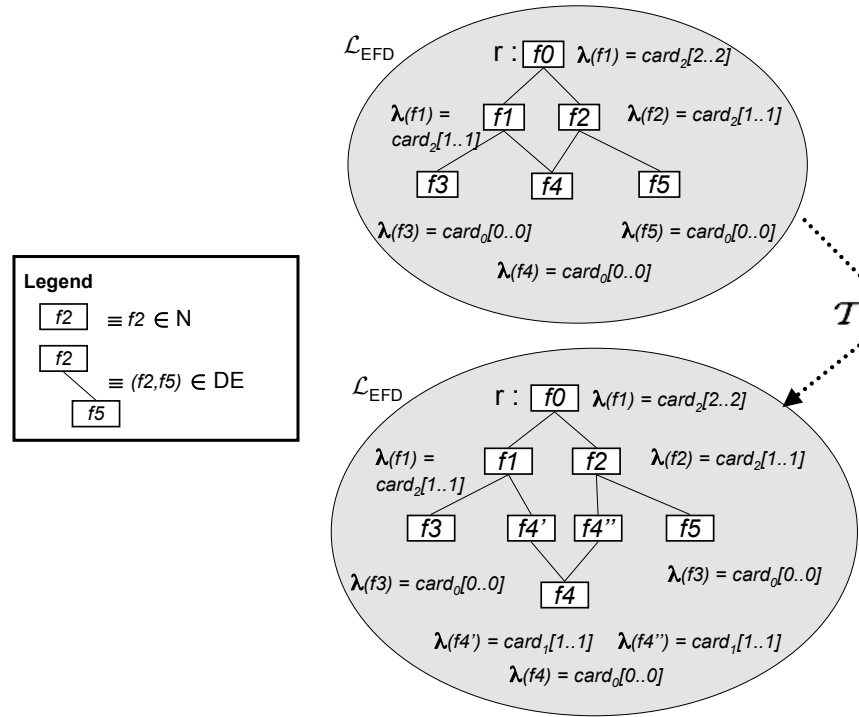
Figure 8.18: From edge-based to node-based Semantics

8.4.2 Trees or DAGs

The second issue concerns the structure of the FDs. Semantically, distinguishing between trees and DAGs brings some interesting consequences. The main FD variants (Kang et al., 1990; Eisenecker and Czarnecki, 2000; Eriksson et al., 2005; Batory, 2005; Czarnecki et al., 2005c) only use trees. This brings several simplifications to our semantics:

1. A textual variant of the language is now easily obtained by using this tree as an abstract syntax tree, see e.g. (van Deursen and Klint, 2002).

2. The justification rule (Definition 8.1.5, last point) simplifies since each node has exactly one parent.

3. We know that each sub-tree contains only primitive features that do not occur in its brothers, since no sharing is allowed. We call this the signature of the sub-tree.

4. The difference between node-based and edge-based semantics vanishes.

5. The semantics of feature trees can now be expressed compositionally: $[\![n]\!]$, where n is a node of a diagram, is the set of products that would be obtained by choosing n as the root of the diagram. Let op_k be the operator of n, and $s_1, \ldots, s_k$ its sons. We observe that

$\llbracket op_k(s_1, \ldots, s_k) \rrbracket = \{\bigcup c_i | \exists v_1, \ldots v_k \in \mathbb{B}.op_k(v_1, \ldots, v_k) = 1 \land \forall i \in 1 \ldots k, (v_i = 0 \Rightarrow c_i = \emptyset) \land (v_i = 1 \Rightarrow c_i \in \llbracket s_i \rrbracket)\}.$

    (a) For *and*-nodes, this simplifies to $\llbracket and_k(s_1, \ldots, s_k) \rrbracket = \{\bigcup_i c_i | c_i \in \llbracket s_i \rrbracket)\}$.

    (b) For *xor*-nodes, this simplifies to $\llbracket xor_k(s_1, \ldots, s_k) \rrbracket = \bigcup_i \llbracket s_i \rrbracket$.

    (c) For *or*-nodes, this simplifies to $\llbracket or_k(s_1, s_2, \ldots, s_k) \rrbracket = \{c_i\} \cup \{\bigcup_i c_i | c_i \in \llbracket s_i \rrbracket)\}$

6. Decision problems on feature tree are less complex than on feature DAGs. For instance, as we will see in Chapter 9, the computational complexity for FD Satisfiability (Section 7.1.1) is NP-Complete for feature DAGs while linear for feature trees.

All these simplifications may convince one to use feature trees rather than feature DAGs. However, as we will see in Section 9.1.2, feature trees without constraints are less expressive than feature DAGs.

8.4.3 Optional Nodes

Optional nodes (hollow circles in concrete syntax and opt_1-nodes in abstract syntax) should be used carefully, especially under *xor*-nodes. Following our semantics, the FD presented in Figure 8.19 admits a product with neither f_2 nor f_3. Indeed, in $\mathcal{L}_{EFD}$, each hollow circle is translated to $card_1[0, 1]$ node (in this case, f_2? and f_3?). Each such node has one son (in this case, respectively f_2 and f_3) that was originally the node under the hollow circle. Therefore, the configurations $\{f_1, f_2?\}$ and $\{f_1, f_3?\}$ are valid since one hollow circle could be selected while its son is not. If the features f_2? and f_3? are non-primitive, then the only resulting product is $\{f_1\}$. This is clearly not the semantics intended in (Riebisch et al., 2002). We simply suggest to use hollow circles with precaution especially under a *xor*-node.

8.4.4 Mandatory Nodes

Similarly to optional nodes, mandatory nodes should be used with precaution. Following our semantics, the FD presented in Figure 8.20 does not admit products with both features "*automatic*" and "*manual*". However, since these features are mandatory it becomes counterintuitive. This ambiguity reappears with the sub-features of the *or*-node "*engine*". In addition, decorating with filled circles the sons of the *and*-node "*car*" is redundant since the *and*-node already guarantees that they will be in every product, if their parent ("*car*") is in the product and if they are not optional.

In our abstract syntax, we prefer to omit mandatory nodes. An example is given in Figure 8.17 where the fact that the mandatory nodes f_1 and f_2 are decorated with a filled circle in EFD does not influence the translation to our FFD abstract syntax.

Following our semantics, determining if a node is mandatory, or not, is only relative to its incoming edge(s). Our justification rule (Definition 8.1.5, last point) says that a node, should it be mandatory or optional, will not be part of a configuration if none of its parents is. This interpretation contradicts the following sentence: *"All mandatory features are part of all [configurations]"*(Streitferdt et al., 2003, p.2). However, in the same paper, the authors seem to agree with our interpretation. Indeed, in Figure 8.21, extracted from (Streitferdt et al., 2003, Figure 3), `Email` is mandatory, but they make clear that it is not included in configurations where `Net` is not present.

EFD Concrete Syntax

$$\mathcal{L}_{EFD} =$$
$$\mathcal{L}_{FFD}(DAG, \{card \cup \{opt_1\}\}, \{\rightarrow, |\}, CR)$$

r : $\boxed{f1}$ $\lambda(f1) = card_2[1..1]$

$\lambda(f2?) = opt_1$ $\boxed{f2?}$ $\boxed{f3?}$ $\lambda(f3?) = opt_1$

$\boxed{f2}$ $\boxed{f3}$

$\lambda(f2) = card_0[0..0]$ $\lambda(f3) = card_0[0..0]$

$\boxed{f1}$

$\boxed{f2}$ 1..1 $\boxed{f3}$

Legend

$\boxed{f2}$ $\equiv f2 \in N$

$\boxed{f2}$
$\quad \equiv (f2,f5) \in DE$
$\boxed{f5}$
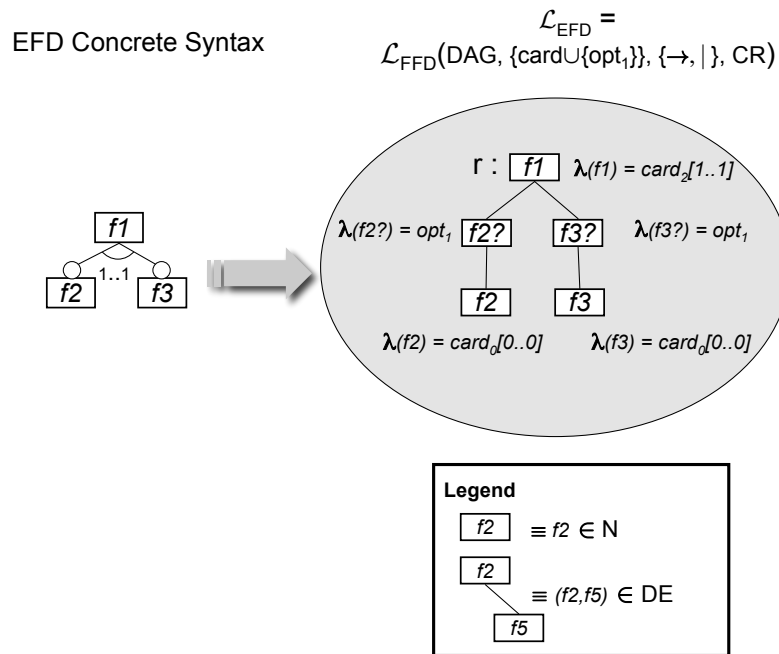
Figure 8.19: Optional Node

car

car body   transmission   engine   pulls trailer

automatic   manual   electric   gasoline
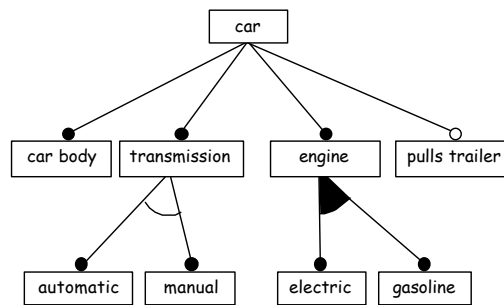
Figure 8.20: Mandatory Node, example from (Eisenecker and Czarnecki, 2000, Figure 4-14)

8.4.5 Semantic Domain

The last issue concerns the definition of the semantic domain (Definition 8.2.2) that is defined as a set of products where a product is a set of primitive nodes $\mathcal{PP}$. The way we define this semantic domain has important consequences:

- Some features are not taken into account to define a product. In FD languages, the atomic building blocks are features (nodes). However, we want to leave the flexibility to the modellers to decide whether a feature is relevant for them to discriminate products and to determine which features are only used for the decomposition. These features should not be taken
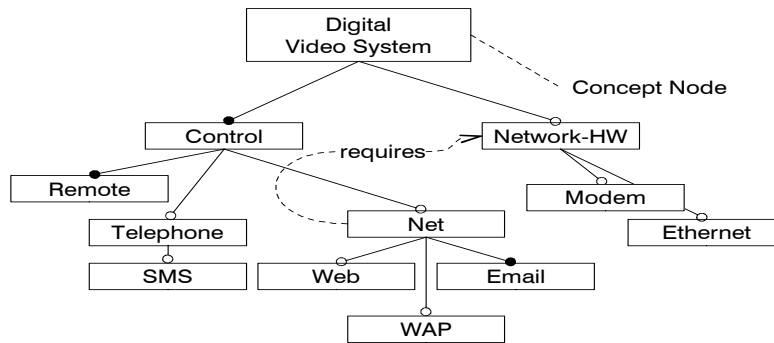
Figure 8.21: EFD example from (Streitferdt et al., 2003, Figure 3)

into account to define a product. Intuitively, we consider that some intermediate features are only used to organise the decomposition of features, while others, the so-called primitive features, are used to effectively distinguish one product from another. Introducing the notions of *primitive* and *non-primitive* nodes (or features) impacts both syntax and semantics. When all nodes are primitive ones ($P = N$), the set of configurations exactly maps the set of products. However, if the FD is a DAG and if non-primitive node(s) exist ($P \subseteq N$) several configurations may map to the same product. When every feature is kept primitive ($P = N$) the original definition of OFD is preserved.

In Figure 8.22, if all nodes are primitive ones ($f_0, \ldots, f_5 \in P = N$) then the resulting products are: $\{f_0, f_1, f_3\}, \{f_0, f_1, f_4\}, \{f_0, f_2, f_4\}$ and $\{f_0, f_2, f_5\}$. On the other hand, if $f_0, f_1, f_2 \in N$ are non-primitive nodes then the resulting products are: $\{f_3\}, \{f_4\}$ and $\{f_5\}$. In this case the configurations $\{f_0, f_1, f_4\}$ and $\{f_0, f_2, f_4\}$ both maps to the same product $\{f_4\}$. This indicates that two different paths exist in the DAG between the root f_0 and the leaf feature f_4. We will see in Chapter 9 that it also worsens computational complexity for some decision problems.
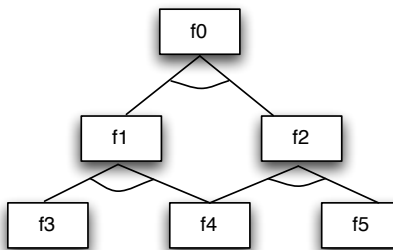
Figure 8.22: Primitive and non-primitive Nodes

- No order is kept in the set of products or in the set of primitive nodes. Intuitively, we consider that two products with the same nodes or two product lines with the same products should be identical. Following our semantic domain, a product $\{f_1, f_2\}$ and a product $\{f_2, f_1\}$ are identical. However, ordering nodes could be relevant. For instance, a node (or feature) may correspond to a transformation (on code or models) (Eisenecker and Czarnecki, 2000; Ryan and Schobbens, 2004) and these transformations may not produce the same result if they are applied in a different order. In this case, an order on primitive nodes should be defined and therefore a product would be defined as a list (van Deursen and Klint, 2002) rather than a set. Since order of features now counts in a product, the number of possible resulting products will grow exponentially.

- Duplication of features is not allowed. Since we define products as sets, a product containing the same feature several times is not allowed. Instead, some authors (Czarnecki et al., 2005b) may prefer to define products as multi-sets. Hence, products $\{f_1, f_1, f_2\}^+$ and $\{f_1, f_2, f_2\}^+$ would be allowed and are different.

8.5 Chapter Summary

In this chapter, we have formalised the informal semantics proposed in (Kang et al., 1990, 1998) and proposed a formal definition for OFD (Kang et al., 1998) following the principles proposed by Harel and Rumpe (Chapter 4). Secondly, we have extended these principles and proposed a formal definition for a family of FD languages (FFD) including OFT (Kang et al., 1990), OFD (Kang et al., 1998), RFD (Griss et al., 1998), GPFT (Eisenecker and Czarnecki, 2000), EFD (Riebisch et al., 2002), VBFD (van Gurp et al., 2001), PFT (Eriksson et al., 2005) and VFD (Bontemps et al., 2004). FFD is configurable and gives a common semantics to the studied informal FD languages. Once FFD is configured, formal criteria such as expressiveness, succinctness and embeddability can be applied. Thirdly, we have discussed semantic issues and solved them according to our own interpretation. We do not claim that our semantic is the right one but at least we have made it explicit. In the next chapter, we will use FFD to compare informal and formal FD languages.

Chapter 9

Feature Diagram Languages: Quality Analysis

In Chapter 3, we have introduced the semiotic quality Framework (SEQUAL) and extended it with formal properties for language evaluation. Throughout this chapter, we apply the method defined in Chapter 7 to evaluate and compare FD languages. First, we compare the informal FD languages covered by FFD (Chapter 8). The main results concerning informal FD languages analysis have been previously published in (Schobbens et al., 2007; Heymans et al., 2008). Then, we compare a first selection of formal languages with FFD. Since formal FD languages are closely related to Propositional Logic, one could question whether other graphical languages associated to Propositional Logic may not replace FDs advantageously. For instance, formal languages such as Boolean Circuits (BC) (Shannon, 1937, 1938; Vollmer, 1999) or Binary Decision Diagrams (BDD) (Akers, 1978) could be considered. To further investigate this question, we propose to compare FFD and Boolean Circuits (Shannon, 1937, 1938; Vollmer, 1999). In addition, we propose to compare FFD with two other formal FD languages supported by tools. In a chronological order, they are respectively van Deursen et *al.*'s language (vDFD) (van Deursen and Klint, 2002) and Batory's language (BFT) (Batory, 2005). The results concerning the analysis of vDFD have been previously published in (Trigaux et al., 2006).

The structure of this chapter is as follows. Firstly, in Section 9.1, we analyse and compare FFD and its members. Then, we investigate its computational complexity (Section 9.1.1), expressiveness (Section 9.1.2), embeddability (Section 9.1.3) and succinctness (Section 9.1.4). These analyses are based on FFD formal definition that has been already provided and extensively discussed in Chapter 8.

Secondly, we analyse and compare formal properties of BCs (Shannon, 1937, 1938; Vollmer, 1999), vDFD (van Deursen and Klint, 2002) and BFT (Batory, 2005) in Sections 9.2, 9.3 and 9.4 respectively. For each of them, we (1) present their formal definition according to Harel and Rumpe's principles (Chapter 4), (2) provide an abstraction function between their semantic domain and FFD semantic domain, (3) study the semantic equivalence or non-equivalence with FFD and (4) analyse their expressiveness, embeddability and succinctness.

9.1   Free Feature Diagram (FFD) Analysis

Following the comparison method described in Figure 7.1 and Section 7.2, once FFD has been formally defined, it guarantees that all members of FFD (1) are in FFD (*X* in FFD?), (2) follow H&R principles (Harel and Rumpe, 2004) (*X* follows H&R?) and (3) share a common semantic domain $PL = \mathcal{PPP}$ ($S_{FFD} = S_X$). When these conditions are satisfied, all FFD members can be analysed and compared according to their formal properties. We start with complexity analysis.

9.1.1   FFD Complexity Analysis

We consider the computational complexity of six fundamental decision problems: Satisfiability (Section 9.1.1.1), Product-Checking (Section 9.1.1.2), Equivalence (Section 9.1.1.3), Intersection (Section 9.1.1.5), Inclusion (Section 9.1.1.4) and Union (Section 9.1.1.6). These decision problems have been already defined in Chapter 7. They rely on the fact that the semantic domain of FFD (S_{FFD}, Definition 8.2.2) is defined with sets.

   In the following sections, we first recall each decision problem and then we discuss their decidability (Is there an algorithm for this problem?) and complexity (What is its relative computational difficulty measured in computation time or memory?).

9.1.1.1   FFD Satisfiability

In terms of FFD, *Satisfiability* is a decision problem that consists in determining if a FD represents a product line that has at least one product (see Definition 9.1.1).

Definition 9.1.1 (FFD Satisfiability) *A FFD diagram d is* satisfiable *iff* $\mathcal{M}_{FFD}[\![d]\!] \neq \emptyset$.

Theorem 9.1.1 *Deciding whether a FD is satisfiable is NP-complete.*

Proof To prove this theorem, we translate the FFD to Existentially Quantified Boolean Formulae (EQBF, Definition 9.1.2) in Conjunctive Normal Form (CNF, Definition 9.1.3) and conversely. Such a formula is satisfiable iff it is satisfiable without the existential quantifiers. This problem is the SAT-problem that is known to be *NP-Complete* (Cook, 1971).

Definition 9.1.2 (Existentially Quantified BF) *Given a set of propositional variables V, the set of Existentially Quantified Boolean Formulae (EQBF) is syntactically defined in BNF notation as:* $EQBF := a \mid \perp \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \oplus \phi_2 \mid \neg\phi_1 \mid \phi_1 \Leftrightarrow \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \exists a \, EQBF \, where$

- *$a \in V$;*

- *$\perp$ is the bottom symbol (FALSE).*

Semantically, $\exists v \, \Sigma \in EQBF$ is TRUE iff ($\Sigma[v/TRUE] \vee \Sigma[v/ \perp]$) is TRUE. $\Sigma[v/v']$ means that in the BF Σ the value of the propositional variable v is set to v'.

   Although it may be exponentially larger, there exists a propositional formula that is equivalent to every EQBF. EQBF satisfiability is $\sum_1 \mathbf{P}$-complete and therefore NP-complete (Papadimitriou, 1994, p.428).

Definition 9.1.3 (BF in Conjunctive Normal Form) *In Propositional Logic, a BF is in Conjunctive Normal Form (CNF) when it is a conjunction of clauses, where a clause is a disjunction of literals.*

- *For easiness*: The FFD semantic function gives a translation from a FFD to EQBF. BF should be existentially quantified to discard non-primitive features.

The translation from FFD to EQBF in CNF consists in:

1. Translate each node in the FD into a *card*-node as defined in Table 9.1.

Instead of ...	write ...
an opt_1-node	a $card_1[0..1]$
a xor_s-node	a $card_s[1..1]$
an or_s-node	a $card_s[1..s]$
an and_s-node	a $card_s[s..s]$

Table 9.1: From nodes to *card*-nodes

2. Translate each *card*-node into CNF. If we consider a node g of type $card_n[i..j]$ with maximum n sons $(f_1 \ldots f_n)$, the translation of this *card*-node into CNF consists in:

   (a) Translate into CNF the *card*-operator (see rule 2 in Definition 8.2.5). This translation (Table 9.2*) is based on the optimal encoding of boolean cardinality in CNF proposed in (Sinz, 2005). This encoding translates boolean cardinality constraints into CNF. Boolean minimal and maximal cardinality constraints (i,j) are formulae expressing that at least i $(\geq i(x_1, ..., x_n))$ and at most j $(\leq j(x_1, ..., x_n))$ out of n propositional variables $(x_1, ..., x_n)$ are TRUE. In (Sinz, 2005), the author proposed two CNF encodings for $\leq j(x_1, ..., x_n)$. The complexity of the encoding is $O(n.j)$ (Sinz, 2005) where n is the number of propositional variables and j the maximal cardinality. This translation and the selection of the encoding are detailed and discussed in Sections 10.3.3.1 and 10.3.3.2.

   (b) Translate into CNF the justification rule (see rule 5 in Definition 8.2.5). The justification rule says that if a node is evaluated to TRUE, at least one of its parent(s) must be too. Except for the root that has no parent and that is always evaluated to TRUE (see rule 1 in Definition 8.2.5). The corresponding translation is given in Table 9.3.

   (c) Make the conjunction of the two BFs resulting from the two previous steps.

3. Translate into CNF each *excludes* or *requires* constraint (see rules 3 and 4 in Definition 8.2.5), would it be graphical or textual. The corresponding translations (the last two rows of Table 9.2) are immediate.

4. Make the conjunctions of the BFs in CNF resulting from the two previous steps.

---

*Formulae in Table 10.1 are not yet in CNF. This is for readability reasons. They are converted to CNF with a standard CNF conversion algorithm.

$\lambda(g)$	Conditions	Conjunctive Normal Form
$card_n[0..0]$	$n \geq 1$	$\bigwedge_{i=1..n}(\neg g \vee \neg f_i)$
$card_n[0..n]$		no output generated
$card_n[0..j]$	$1 \leq j < n, n \geq 2$	$\neg g \vee \leq j(f_1, \ldots, f_n))$
$card_n[1..n]$	$n \geq 1$	$\neg g \vee f_1 \vee \ldots \vee f_n$
$card_n[n..n]$	$n \geq 2$	$\bigwedge_{i=1..n}(\neg g \vee f_i)$
$card_n[i..j]$	$1 \leq i \leq j < n, n \geq 2$	$\neg g \vee (\geq i(f_1, \ldots, f_n) \wedge \leq j(f_1, \ldots, f_n))$
f_1 requires f_2		$\neg f_1 \vee f_2$
f_1 excludes f_2		$\neg f_1 \vee \neg f_2$

Table 9.2: FFD to CNF

VFD Nodes	Justification rule in CNF
$card_n[i..j] = root$	g
$card_n[i..j] \neq root$	$\neg g \vee p_1 \vee \ldots \vee p_m$

Table 9.3: Boolean Formula for justification rule

5. Existentially quantified non-primitive features in the BF. Once non-primitive features are allowed, the BFs should be existentially quantified with non-primitive features. Indeed, when variables are existentially quantified within a BF, they do not influence its satisfiability (see last column in Table 9.4). These variables will correspond to the non-primitive features. Fortunately, when there is no alternation of quantifiers within a BF, its satisfiability remains $\sum_1$ **P**-*complete* and hence *NP-complete* (Papadimitriou, 1994, p. 428). However, as we will see further, this is not the case for equivalence (Section 9.1.1.3) and inclusion (Section 9.1.1.4) for which there is an alternation of quantifiers.

x	y	$x \vee y$	$x \wedge y$	$\exists x \, x \vee y$	$\exists x \, x \wedge y$
1	1	1	1	1	1
1	0	1	0	1	0
0	1	1	0	1	1
0	0	0	0	1	0

Table 9.4: Existentially Quantified Boolean Formula

Example 9.1.1 *As an example, we illustrate in Table 9.5 the translation of a FD into EQBF. This FD is composed of three nodes and two edges. The root of the FD is a non-primitive node n of node type $card_2[1..2]$. This root is decomposed in two sons (f_1, f_2) that are primitive nodes ($f_1, f_2 \in P$) of type and_0. This FD is then translated into an EQBF. The node n is translated according to the fourth line of Table 9.2 and the first line of Table 9.3. In addition, r is the root and is non-primitive. Hence, r must be always evaluated to TRUE and existentially quantified. The nodes f_1, f_2 are translated according to the second line of Table 9.2 and the*

second line of Table 9.3. We notice that only their justification rules matter since they are of type $and_0 = card_0[0..0]$.

Instead of . . .	write . . .
	$\exists n$ $n \wedge$ $(\neg n \vee f_1 \vee f_2) \wedge$ $(\neg f_1 \vee n) \wedge$ $(\neg f_2 \vee n)$

Table 9.5: Example: a FD into EQBF

- *For hardness*: Checking whether an EQBF in a CNF is satisfiable is known to be *NP-complete* (Papadimitriou, 1994, p. 428). To prove that at worst satisfiability checking for FD language is *NP-complete*, we need to provide a translation from EQBF in CNF to each member of FFD. The easiest way is to define two translations in polynomial time:

  1. The first one translates any EQBF (in CNF) into a simplified version of RFD that we call Constraintless Feature RSEB Diagrams language (CRFD) (Figure 9.1). This simplification removes constraints in RFD.

     An EQBF is in CNF when it is of the form $\exists p_x \ldots p_y \, C_1 \wedge \ldots \wedge C_i \wedge \ldots \wedge C_n$ where:

     - $p_x \neq p_y$;
     - all C_i are clauses of the form $C_i = (p_1^i \vee \ldots \vee p_j^i \vee \ldots \vee p_m^i)$;
     - p_j^i are literals of the form p_j (positive literal) or $\neg p_j$ (negative literal);
     - p_j is a propositional variable;

     As illustrated in Figure 9.1, the translation from EQBF to CRFD ($T_{EQBF \rightarrow CRFD}$) consists in:

     - For each p_j, create a *xor*-node (Xp_j) related to the root r with two sons: the positive literal (p_j) and its negation ($\neg p_j$).
       The negation of a node n, written $\neg n$, is translated into a *xor₂*-node with two sons: the original node n and the TRUE node t. The node t is always evaluated to TRUE and corresponds to an *and₀*-node (i.e with no son) that is directly related to the root. As t is always TRUE, $\neg n$ is only TRUE when n is FALSE. For abbreviation, the negative node of n is named $\neg n$ but the reader should keep in mind that it stands for a more complex diagram as illustrated in Table 9.6.
     - For each clause ($C_i = p_1^i \vee \ldots \vee p_j^i \vee \ldots \vee p_m^i$), create an *or*-node ($C_i$) and decompose it into one node for each of its literals. These nodes have already been created in the previous point. The purpose here is to connect the node (C_i) with its correct literals;
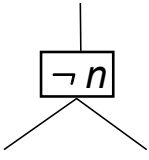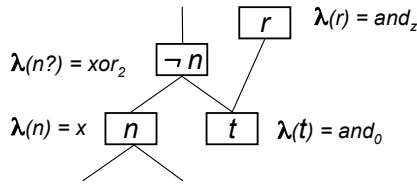
Instead of ...	write ...

Table 9.6: Translation: Node Negation

- Add a root (r) as an *and*-node connected to all *or*-nodes (corresponding to clauses) and all *xor*-nodes (corresponding to literals);
- Finally, each existentially quantified variable ($p_x, \ldots, p_y$) is set as a non-primitive node within the CRFD ($p_x, \ldots, p_y \notin P$). In addition, auxiliary nodes (X_{p_j}, $\neg p_j$, C_i, t and r with $1 \le j \le m$ and $1 \le i \le n$) introduced by the translation are also set as non-primitive nodes. The other nodes (variables that are not existentially quantified) are set as primitive nodes.
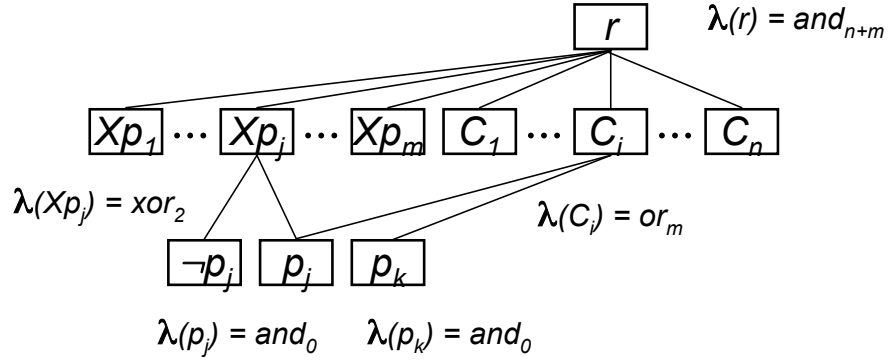
Figure 9.1: Translation: EQBF into CRFD

One could argue that this translation forgets to take into account the justification rules added for each node contained in the resulting FD. Indeed, the justification rule of the node p_j says that at least one parent of p_j should be evaluated to TRUE. However, X_{p_j} is always TRUE since X_{p_j} is an *and*-son of the root r that is always TRUE. The implementation of the semantics of the FD given in Figure 9.1 produces a BF in which clauses corresponding to the justification rules can be eliminated.

- Since r is a root and thus is always TRUE, every justification rule of the form $s_i \Rightarrow r$ associated to sons (s_i) of the *and*-root r can be eliminated. Therefore, every justification rule for the nodes X_{p_j}, $1 \le j \le m$ and C_i, $1 \le i \le n$ can be eliminated.
- Since r is an *and*-root, all its sons are also TRUE and the justification rule associated to the sons of the sons of r are also redundant. For instance, the justification rule

for p_j says that $p_j \Rightarrow X_{p_j} \vee C_i \vee \neg p_j$. We know that r is always true and, since r is an *and*-node, that $r \Rightarrow X_{p_j} \wedge Ci$. Hence, X_{p_j} and Ci are true and the right part of the implication $(X_{p_j} \vee C_i \vee \neg p_j)$ is always TRUE. This means that the value of p_j does not matter here and that the justification rule $p_j \Rightarrow X_{p_j} \vee C_i \vee \neg p_j$ can be eliminated.

- p_j is also shared by its negation, the node "$\neg p_j$", as illustrated in Figure 9.2. In addition, the node "$\neg p_j$" is evaluated to FALSE when p_j is selected. The resulting BF for this FD is: $r \wedge (r \Rightarrow X_{p_j} \wedge t) \wedge (X_{p_j} \Rightarrow \text{“}\neg p_j\text{”} \oplus p_j) \wedge (\text{“}\neg p_j\text{”} \Rightarrow p_j \oplus t)$. Four justification rules are added (by conjunction):
  * $t \Rightarrow \neg p_j \vee r$;
  * $X_{p_j} \Rightarrow r$;
  * $p_j \Rightarrow X_{p_j} \vee \neg p_j \vee C_i \ldots$;
  * $\neg p_j \Rightarrow X_{p_j}$.

  We already know that all the right parts of these implications are TRUE therefore they can be eliminated.
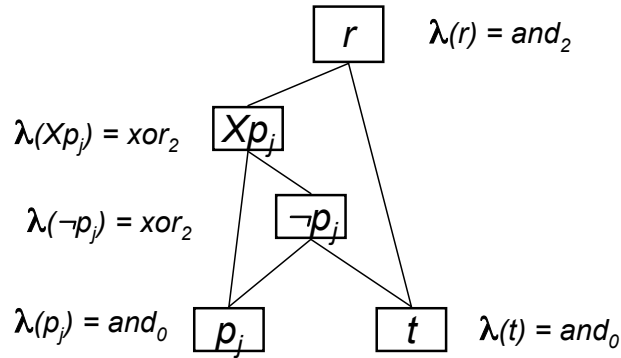
Figure 9.2: Justification rule elimination

2. For the second translation we show how CRFD can be translated into COFD. COFD is the simplified version of OFD to which we have removed constraints. Similar translations exist with all FD languages included in FFD as we will see in the analysis of the succinctness of FFD (Section 9.1.4).

   As defined in Table 9.7 the translation from CRFD to COFD consists in translating *or*-nodes that are allowed in CRFD but not in COFD. An *or*-node n with m sons f_i is translated into a *xor*-node with m sons. Each of its sons s_i is an *and*-node with a son f_i and the other $m - 1$ are optional nodes f_j, $j \neq i$.

$\square$

The above complexity result (Theorem 9.1.1) seems rather pessimistic. However, it should be mitigated. For instance, in one specific case, checking satisfiability for FDs is linear and, in other cases, it is even constant.

Instead of ...	**write ...**

The left cell shows a diagram: node n with $\lambda(n) = or_m$, with sons $f_1 \cdots f_i \cdots f_m$, and $\lambda(f_i) = x$.
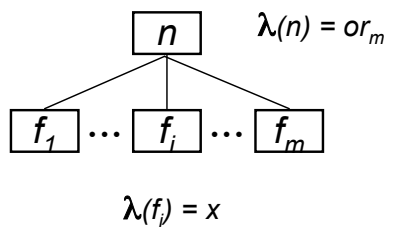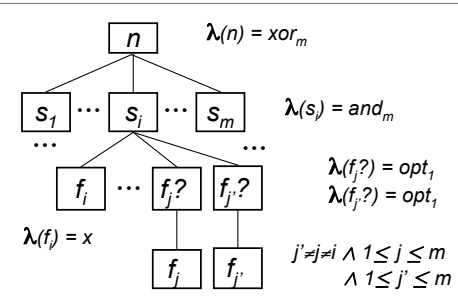
The right cell shows a diagram: node n with $\lambda(n) = xor_m$; sons $s_1 \cdots s_i \cdots s_m$ with $\lambda(s_i) = and_m$; under s_i are $f_i \cdots f_j? \; f_{j'}?$ with $\lambda(f_j?) = opt_1$, $\lambda(f_{j'}?) = opt_1$, $\lambda(f_i) = x$; and leaves $f_j \; f_{j'}$ with $j' \neq j \neq i \wedge 1 \leq j \leq m \wedge 1 \leq j' \leq m$.

Table 9.7: Embedding: CRFD to COFD

1. For FTs without constraints satisfiability is checked in linear time. We propose the following algorithm:

   (a) The algorithm first eliminates inconsistent *card*-nodes (Definition 9.1.4), replaces them by xor_0 representing FALSE. A *card*-node whose lower bound is above the number of sons is translated into xor_0;

   Definition 9.1.4 (Consistent and inconsistent *card*-node) *Consistent card-node are defined as $card_s[i..j]$ where $0 \leq i \leq j$. Otherwise they are inconsistent.*

   (b) Then the algorithm propagates all FALSE node (xor_0) up in the tree. For each xor_0:
   - If its parent is a *xor*-node, an *opt*-node or an *or*-node then remove xor_0;
   - If its parent is an *and*-node, then remove xor_0 and translate the parent into xor_0;
   - If its parent is a *card*-node, then remove xor_0 and check if the number of sons of the parent is still over or equal to its lower bound. If it is not the case, translate the parent into xor_0;

   (c) The propagation either brings a xor_0 to the root and the tree is unsatisfiable, or eliminates all xor_0 from the tree that is satisfiable. If $\lambda(root) = xor_0$ then return "no", else return "yes".

2. For FDs with only *xor*-nodes except for the leaves that can be and_0-nodes, satisfiability is checked in constant time. The algorithm checks whether there exists and_0-node (to exclude the case xor_0 as root). Each and_0-node corresponds to one product;

3. For FDs with only *and*-nodes, satisfiability is checked in constant time. Such FDs are always satisfiable since only one product always exists. This product is either composed of all the primitive features or of none (The case when no primitive features have been defined). Hence the algorithm always returns "yes";

4. For FDs in Normal Form (Definition 9.1.5), satisfiability is checked in constant time. The algorithm checks whether at least one product exists and therefore checks whether the root has at least a son or not.

Definition 9.1.5 (Normal Form for a FFD) *A FFD is in a* Normal Form *when*

- *it is a DAG;*

- *its root is the only xor-node and this node is non-primitive;*

- *each product is directly connected to the root and is represented by a non-primitive and-node with sons corresponding to the constituent features of the product. Therefore, the product with no feature is represented by and_0-node without son;*

- *it does not possess any opt-, or-, card-node or constraint.*

$\lambda(r) = xor_3$

$\lambda(p1) = and_1$

$\lambda(p3) = and_1$

$\lambda(p2) = and_2$

$\lambda(fi) = and_0$
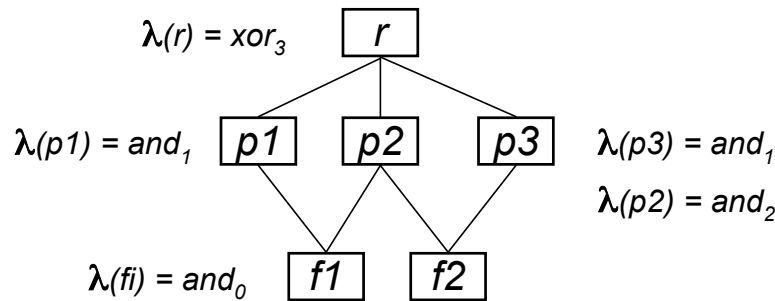
Figure 9.3: FFD Normal Form: FD of product line $\{\{f_1\}, \{f_2\}, \{f_1, f_2\}\}$

Table 9.8 summarises the complexity results obtained for satisfiability. The first row corresponds to the most general case: a FD language on which no restriction is imposed. The last row corresponds to a specific case: a FD language where only *and*-nodes are allowed. In addition, modellers do not necessarily use all the constructs offered by a FD language. The table could also be used to study complexity when checking the satisfiability of one particular model containing a specific subset of constructs.

Translating a propositional formula to FDs without constraints or sharing is mostly impossible because such FDs are not expressively complete. The satisfiability problem is *NP-Complete* for all "useful" FD languages, where useful means expressively complete (Definition 5.2.2). Note that any FD can be simply translated into a BF. It suffices to encode the FD constraints in Propositional Logic. It can then be handed to some SAT-solvers to check whether it is satisfiable. Such solvers can efficiently check formulae of thousands of variables, and could thus be practically very helpful here.

TREE/DAG	AND	XOR	OR	OPT	Card	Requires, Excludes	Satisfiability
DAG							*NP-Complete*
TREE						$\sqrt{}$	*NP-Complete*
TREE				$\sqrt{}$		$\times$	Linear
DAG	$\sqrt{}$	$\times$	$\times$	$\times$	$\times$	$\times$	Constant

Table 9.8: Satisfiability: Complexity Results

9.1.1.2   FFD Product-Checking

Product-checking verifies whether a given product (set of primitive features) is in the product line of a FD (see Definition 9.1.6). This is not as trivial as expected (Theorem 9.1.2).

Definition 9.1.6 (FFD Product-Checking) *A product p is in the product line of a FFD d iff $p \in \mathcal{M}_{FFD}[\![d]\!]$.*

Theorem 9.1.2 *Product-checking is NP-complete.*

Proof To prove this theorem, we translate the FFD to EQBF and conversely.

- *For easiness*: The FFD is translated into its equivalent EQBF Σ_1 according to the translation provided in Theorem 9.1.1. Then, this formula is simplified according to the known truth value of its primitive nodes. It remains to discover a valuation for the non-primitive nodes, that can be solved by SAT. This is why product-checking is not linear for FFD. Actually, the Boolean values corresponding to non-primitive features will not be bound in the logical expression and therefore all possible configurations should be computed. Therefore, product-checking is a SAT-problem that is *NP-complete*.

  Example 9.1.2 *Once, in Figure 8.22, f_0, f_1 and f_2 are non-primitive features and f_3, f_4 and f_5 are primitive ones, the Boolean values for f_0, f_1 and f_2 are not bound while they are, according to the product to check, for f_3, f_4 and f_5. Therefore, checking the membership of the product $\{f_3, f_4\}$ consists in checking the satisfiability of the following logical expression where f_3 and f_4 are bound to TRUE while f_5 is bound to FALSE:*
  $(f_0 \implies xor(f_1, f_2)) \wedge (f_1 \implies xor(f_3, f_4)) \wedge (f_2 \implies xor(f_4, f_5)) \wedge f_0 \wedge (f_1 \implies f_0) \wedge (f_2 \implies f_0) \wedge (f_3 \implies f_1) \wedge (f_5 \implies f_2) \wedge (f_4 \implies (f_1 \vee f_2))$
  In the end, the satisfiability result for product $\{f_3, f_4\}$ is "no".

- *For hardness:* The EQBF in CNF corresponding to the SAT problem can be translated to an RFD as already presented in Theorem 9.1.1. Then, we need to construct the product based on the values given for the non existentially quantified variables of the EQBF. Each non existentially quantified variable that is set to TRUE is a primitive feature included in the product to check.

$\square$

Here also, the complexity result should be mitigated. Product-checking for some specific FD languages is easier. Indeed, in several cases, product-checking can be computed in linear time:

1. FDs where all nodes are primitive. Here, the truth values of all nodes will be set in the propositional formula. The value of this formula is then computed in linear time.

   Example 9.1.3 *Once all the features in Figure 8.22 are primitive ones, their Boolean values should be bound in the corresponding logical expression. Indeed, all the primitive features in the product ($\{f_0, f_1, f_2, f_3, f_5\}$) to check will be set to TRUE while the others to FALSE. Hence, f_0, f_1, f_2, f_3, f_5 are bound to TRUE while f_4 is bound to FALSE in the logical expression given in Example 9.1.2. In the end, the satisfiability result for product $\{f_0, f_1, f_2, f_3, f_5\}$ is "yes" and is obtained in linear time.*

2. FTs with *and-*, *xor-*, *card-*, *or-*, *opt*-nodes where all and_0 and $card_0[0..j]$ leaves are primitive. The algorithm marks the tree with truth values, starting from the primitive leaves. Leaves that are or_0, xor_0 or $card_0[i..j]$ with $i > 0$ can be marked as FALSE. The other leaves are marked with the value of their primitive features. Successively the parents of the primitive leaves are labelled until the root is reached following these rules:

   - When a primitive non-leaf node is reached, the algorithm checks whether the computed value is the value in the product and if so, labels this node with TRUE; otherwise the algorithm stops and answers "no".
   - When a non-primitive node is reached, the algorithm labels the node with its computed value.

   In the end, if the root is labelled TRUE, the algorithm answers "yes"; otherwise "no".

3. FDs with only *xor*-nodes except the leaves that are and_0-nodes. If the product contains more than one primitive leave then the algorithm answers "no". Otherwise, if all primitive nodes of the product are on the same path in the DAG from the root until the primitive leave, the algorithm answers "yes"; otherwise "no".

4. FDs with only *and*-nodes. The algorithm checks that the given product is the set of primitive nodes reachable from the root.

5. FDs in the Normal Form (Definition 9.1.5). The algorithm checks if there exists an *and*-son of the root for which each sub-feature is a feature included in the product to check.

Table 9.9 summarises the complexity results obtained for product checking. We mainly notice that when all nodes are primitive ones ($P = N$), product-checking complexity decreases drastically from *NP-Complete* to Linear (see row 4 in Table 9.9) .

TREE/DAG	AND	XOR	OR	OPT	Card	Primitive Nodes	Requires, Excludes	Product-Checking
DAG						$P \subset N$		NP-Complete
TREE						$P \subset N$	$\sqrt{}$	NP-Complete
TREE						and_0 and $card_0[0..j]$ leaves $\subseteq P \subset N$	$\times$	Linear
DAG						$P = N$		Linear
DAG	$\sqrt{}$	$\times$	$\times$	$\times$	$\times$		$\times$	Linear

Table 9.9: Product-Checking: Complexity Results

9.1.1.3 FFD Equivalence

Equivalence of two FFDs is needed whenever we want to compare two versions of a product line (for instance, after a refactoring). When they are not equivalent, the algorithm can exhibit a product showing their difference (see Definition 9.1.7).

Definition 9.1.7 (FFD Equivalence) *Two FFDs d_1 and d_2 are* equivalent *iff* $\mathcal{M}_{FFD}[\![d_1]\!] = \mathcal{M}_{FFD}[\![d_2]\!]$.

Theorem 9.1.3 *Deciding whether two FFDs are equivalent is $coNP^{NP}$-complete. Equivalent means that they share the same product (Definition 9.1.7).*

Proof To prove this theorem, we map it to a $coNP^{NP}$ problem, that is, $coNP^{NP}$-SAT. This problem checks whether a Quantified Boolean Formula with one alternation of quantifier is satisfiable.

- *For easiness*: FFDs d_1 and d_2 are translated into the EQBF Σ_1 and Σ_2 respectively quantified existentially with non-primitive features (N_1 and N_2) that are supposed to be disjoint. Thus equivalence is logical equivalence of two EQBFs that is solvable by $coNP^{NP}$-SAT. Indeed, the equivalence of two EQBFs corresponds to two implications:
  - The first one says that for each product p we are assured that when there are non-primitive features (N_1) that satisfy Σ_1, then there are non-primitive features (N_2) that satisfy Σ_2. $\forall p (\exists N_1 \Sigma_1) \Rightarrow (\exists N_2 \Sigma_2)$.
  - Conversely, the second implication says that for each product p we are assured that when there exists non-primitive features (N_2) that satisfies Σ_2 then there exists non-primitive features (N_1) that satisfies Σ_1. $\forall p (\exists N_2 \Sigma_2) \Rightarrow (\exists N_1 \Sigma_1)$.

  The simplification of these two implications gives: $\forall p \forall N_1 \exists N_2 (\Sigma_1 \Rightarrow \Sigma_2)$ and $\forall p \forall N_2 \exists N_1 (\Sigma_2 \Rightarrow \Sigma_1)$. This formula includes one alternation of quantifier, hence its satisfiability is $\Pi_2\mathbf{P}$-*complete* (Papadimitriou, 1994, p. 428) and logical equivalence of two EQBFs is $\Pi_2\mathbf{P}$-*complete* or $coNP^{NP}$-complete.

- *For hardness*: Conversely, a $coNP^{NP}$-SAT problem of the form $\forall X_1 \exists X_2 \Sigma$ is amenable to an equivalence of RFDs with Σ of the form $(C_1 \wedge \ldots \wedge C_i \wedge \ldots \wedge C_n)$ where each C_i is a clause $(C_i = p_1^i \vee \ldots \vee p_j^i \vee \ldots \vee p_m^i)$ and each p_j^i is a literal. First we apply on $\exists X_2 \Sigma$ the translation $T_{EQBF \to CRFD}$ (Figure 9.1) to obtain the corresponding CRFD. This translation introduces auxiliary nodes $X_{p_j}, \neg p_j, C_i, t, r$ ($1 \le i \le n$ and $1 \le j \le m$) that are non-primitive features and should be existentially quantified. Hence, $\exists X_2 \Sigma \equiv \mathcal{M}_{FFD}[\![T_{EQBF \to CRFD}(\Sigma)]\!]$ with X_2, X_{p_j}, $\neg p_j$, t and r set as non-primitive features.

  Accordingly, $\forall X_1 \exists X_2 \Sigma = \forall X_1 \mathcal{M}_{FFD}[\![T_{EQBF \to CRFD}(\Sigma)]\!]$. Finally, it remains to check the equivalence between the resulting RFD ($T_{EQBF \to CRFD}(\Sigma)$) and the RFD that allows all possible combinations of X_1. This last RFD corresponds to an *or*-root related to all literals contained in X_1, that are all optional features.

  □

Here again, this complexity result should be mitigated (see Table 9.10). Equivalence for some specific FD languages is easier. The equivalence complexity is reduced when all nodes are primitive ones ($P = N$). In this situation, equivalence between FDs corresponds to equivalence of their valid configurations instead of their products. Indeed, when $P = N$, equivalence is *coNP-Complete* (Theorem 9.1.4).

Theorem 9.1.4 *Deciding whether two FFDs d_1 and d_2 have the same valid configurations is coNP-complete.*

Proof To prove this theorem we translate both FFDs to BFs and conversely.

- *For easiness*: d_1 and d_2 are translated into their non quantified formula equivalents Σ_1 and Σ_2, respectively, according to the translation provided in Theorem 9.1.1. However, the last step in the translation, that existentially quantify non-prinmitive nodes, is not applied. Then, we check their equivalence ($\Sigma_1 \equiv \Sigma_2$) that is known to be coNP-complete for BF (Papadimitriou, 1994).

- *For hardness*: Both BFs Σ_1 and Σ_2 in CNF can be translated to two RFDs (r_1, r_2) following the translation presented in Theorem 9.1.1. They are equivalent iff $\mathcal{M}_{FFD}[\![r_1]\!] = \mathcal{M}_{FFD}[\![r_2]\!]$.

  □

In addition, equivalence for FDs can be checked in linear time in other cases:

1. FTs with *and-*, *xor-*, *card-*, *or-*, *opt*-nodes and $P = N$. The algorithm translates every node into a *card*-node following the translation provided in Table 9.1 and eliminates the inconsistent *card*-nodes (Definition 9.1.4). Then it checks whether the trees are the same. If they are not sharing the same *card*-nodes or edges, either a *card*-node has a different cardinality and we can construct different products, or it has different sons and again we can construct a product expressing this difference.

2. FDs with only *xor*-nodes except for the leaves that can be *and*$_0$-nodes. The algorithm eliminates *xor*$_0$-nodes and unreachable primitive nodes and checks that reachable primitive nodes are the same;

3. FDs with only *and*-nodes. We know that all nodes are reachable. The algorithm simply checks that d_1 and d_2 have the same primitive features.

4. FDs in the Normal Form defined in Definition 9.1.5. The algorithm checks that d_1 and d_2 are the same. Every product cited in the list under the root of d_1 is cited in the list under the root of d_2 and conversely.

Table 9.10 summarises the complexity results obtained for equivalence of FDs. Once a FD language distinguishes primitive and non-primitive nodes complexity increases drastically.

TREE/DAG	AND	XOR	OR	OPT	Card	Requires, Excludes	Primitive Nodes	Equivalence
DAG							$P \subset N$	$coNP^{NP}$-Complete
DAG							$P = N$	$coNP$-Complete
TREE						√	$P \subset N$	$coNP^{NP}$-Complete
TREE						√	$P = N$	$coNP$-Complete
TREE						×	$P = N$	Linear
DAG	√	×	×	×	×	×		Linear

Table 9.10: Equivalence: Complexity Results

9.1.1.4   FFD Inclusion

Inclusion (Definition 9.1.8) is useful to check whether a product line is included into another one. For instance, it allows checking whether a new product line takes into account all the products contained in the previous one.

Definition 9.1.8 (FFD Inclusion) *A FFD d_1 is included into a FFD d_2 iff $\mathcal{M}_{FFD}[\![d_1]\!] \subseteq \mathcal{M}_{FFD}[\![d_2]\!]$.*

Theorem 9.1.5 *Deciding whether a FFD d_1 is included into a FFD d_2 is $coNP^{NP}$-complete.*

Proof The proof is similar to the one provided for equivalence except that FFD inclusion is logical inclusion of two EQBFs that is also solvable by $coNP^{NP}$-SAT. Indeed, the inclusion of two EQBF corresponds to one implication that says: for each product p when there are non-primitive features (N_1) that satisfy Σ_1, then there are non-primitive features (N_2) that satisfy Σ_2. Formally: $\forall p(\exists N_1 \Sigma_1) \Rightarrow (\exists N_2 \Sigma_2)$. This implication can be transformed into $\forall p \forall N_1 \exists N_2 (\Sigma_1 \Rightarrow \Sigma_2)$ with only one alternation of quantifiers. Hence its satisfiability is $\Pi_2 \mathbf{P}$-*complete* (Papadimitriou, 1994, p. 428) and logical inclusion of two EQBFs (Definition 9.1.2) is $\Pi_2 \mathbf{P}$-*complete* or $coNP^{NP}$-complete.

□

Here again, this complexity result should be mitigated. Indeed, inclusion for FDs can be checked in linear time in other cases:

1. FTs with *and-*, *xor-*, *card-*, *or-*, *opt-*nodes and $P = N$. The algorithm translates every node into a *card*-node following the translation provided in Table 9.1 and eliminates the inconsistent *card*-nodes (Definition 9.1.4). Then it checks whether d_1 shares with d_2 the same *card*-nodes and edges, either a *card*-node has a different cardinality and we can construct different products, or it has different sons and again we can construct a product expressing this difference.

2. FDs with only *xor*-nodes except for the leaves that can be and_0-nodes. The algorithm eliminates xor_0-nodes and unreachable primitive nodes and checks that reachable primitive nodes of d_1 are included in the one of d_2;

3. FDs with only *and*-nodes. We know that all nodes are reachable and that the PL contains only one product. The algorithm simply checks that d_1 and d_2 have the same primitive features.

4. FDs in the Normal Form defined in Definition 9.1.5. Every product cited in the list under the root of d_1 is cited in the list under the root of d_2.

The complexity results (Table 9.10) obtained for equivalence can be transposed to inclusion.

9.1.1.5 FFD Intersection

Intersection is useful when two feature interference engineers work independently and therefore obtain two different restrictions of the initial product line. To put their work together, we must produce a FFD representing the intersection of two given FFDs (see Definition 9.1.9).

Definition 9.1.9 (FFD Intersection) *A FFD d_3 is the* intersection *between FFDs d_1 and d_2 iff* $\mathcal{M}_{FFD}[\![d_3]\!] = \mathcal{M}_{FFD}[\![d_1]\!] \cap \mathcal{M}_{FFD}[\![d_2]\!]$

Theorem 9.1.6 *A FFD expressing the intersection of two given FFDs (d_1, d_2) can be computed in linear size and time.*

Proof The algorithm to compute the intersection of two given FFDs (d_1, d_2) is the following (see Figure 9.4):

1. For each primitive node in d_1 that is also a node in d_2 (f_1, f_2) prime it in d_2 (f'_1, f'_2), so that they are disjoint in d_1, d_2;

2. For each primitive node in d_1 that is not in d_2 (f_3, f_4) create a fresh identical node but that is primed. Join these new nodes (f'_3, f'_4) and the root of d_2 (r_2) by a fresh *xor*-node ($x2$);

3. Similarly, for each primitive node in d_2 that is not in d_1 (f_5, f_6) create a fresh identical node but that is primed. Join these new nodes (f'_5, f'_6) and the root of d_1 (r_1) by a fresh *xor*-node ($x1$);

4. Set all the prime nodes as non-primitive nodes;

5. Unify d_1 and d_2 (we mean the union of nodes, edges and constraints, respectively);

6. Join x_1, x_2 by a fresh *and$_2$*-node r that is the new non-primitive root;

7. For each primitive feature p that has been primed, we add two constraints p *requires* p' and p' *requires* p.
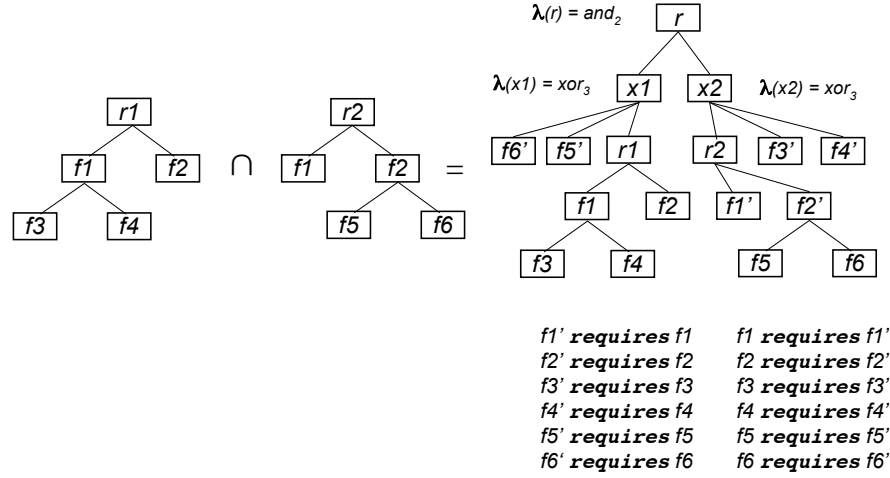
Figure 9.4: FFD: Intersection

□

9.1.1.6   FFD Union

Union of FDs (see Definition 9.1.10) is useful when teams validate in parallel the feature combinations that lead to an acceptable product, without feature interference. Their work can be recorded in separate FDs. The union of these FDs will represent the validated products. For FD languages based on DAGs, this problem is solved in linear time, but the resulting FD should probably be simplified for readability.

Definition 9.1.10 (FFD Union) *A FFD d_3 is the* union *between FFDs d_1 and d_2 iff $\mathcal{M}_{FFD}[\![d_3]\!] = \mathcal{M}_{FFD}[\![d_1]\!] \bigcup \mathcal{M}_{FFD}[\![d_2]\!]$*

Theorem 9.1.7 *A FFD for the disjunction (or union) of two given FFDs d_1 and d_2 can be computed in linear size and time.*

Proof The algorithm to compute the union of two given FFDs d_1, d_2 is the following (see Figure 9.5):

1. Unify d_1 and d_2 sharing primitive features (f_1, f_2) (we mean the union of nodes, edges and constraints, respectively);

2. Join their roots r_1, r_2 by a fresh xor_2-node r that is the new non-primitive root.
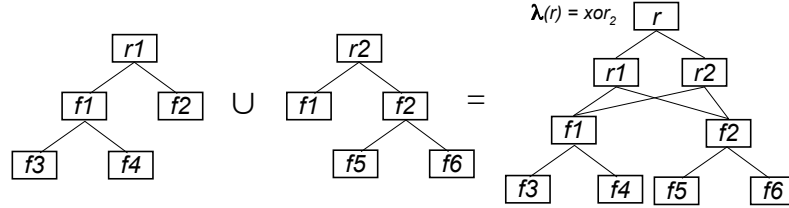
Figure 9.5: FFD: Union

$\square$

This disjunction is computed on the set of products and does not generate new products. It simply makes the union of the set of products represented by d_1 and the set of products represented by d_2. One other interesting problem would be to compute disjunction on the product themselves. This will generate new products including the primitive features of d_1 and d_2 and respecting the constraints imposed by d_1 and d_2. This is obtained by a reduced product:

Definition 9.1.11 *The* reduced product *of d_1, d_2, noted $d_1 \times d_2$, is $\{p_1 \cup p_2 | p_1 \in \mathcal{M}_{FFD}[\![d_1]\!], p_2 \in \mathcal{M}_{FFD}[\![d2]\!]\}$.*

Theorem 9.1.8 *A FFD for the reduced product of d_1, d_2 can be computed in linear size and time.*

Proof The algorithm to compute the reduced product of two given FFDs d_1, d_2 is the following (see Figure 9.6):

1. Unify d_1 and d_2 sharing primitive features (f_1, f_2) (we mean the union of nodes, edges and constraints, respectively);

2. Join their roots r_1, r_2 by a fresh and_2-node r that is the new non-primitive root.

$\square$
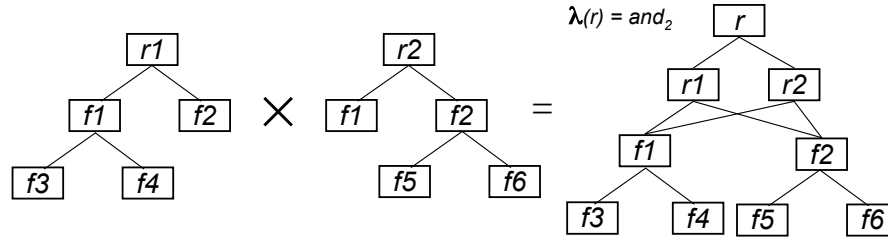
Figure 9.6: FFD: Reduced product

9.1.2   FFD Expressiveness Analysis

Intuitively, the *expressiveness* (Definition 5.2.1) of a language is the part of its semantic domain that it can express. For FD languages expressiveness, the distinction between FD languages that only admit trees and the ones that allow sharing of features by more than one parent turns out to be important. In addition, it is also important to study how the expressiveness of FD languages evolves when some constructs are removed. For instance, we will see that the expressiveness results completely change when OFT does not include a constraint language (Compare Theorems 9.1.10 and 9.1.9). This sub-language of OFT is called Constraintless Original Feature Trees (COFT). Other sub-languages will be used in the sequel. We gather in Table 9.11 their definitions according to FFD where *OP* is a parameter for the type of Boolean operators allowed in *NT* (see the five last entries in Table 9.11):

- Constraintless Feature Diagrams with $NT = OP$ (CFD(OP)),

- Constraintless Original Feature Diagrams (COFD),

- Constraintless Feature RSEB Diagrams (CRFD),

- Constraintless Feature Trees with $NT = OP$ (CFT(OP)),

- Constraintless Original Feature Trees (COFT),

- Constraintless Feature RSEB Trees (CRFT).

9.1.2.1   Tree variants

The main FD variants (Kang et al., 1990; Eisenecker and Czarnecki, 2000; Eriksson et al., 2005; Batory, 2005; van Deursen and Klint, 2002; Czarnecki et al., 2005c) use trees only, instead of single rooted DAGs. We called them "Feature Trees" (FTs). One readily observes that FTs without constraints are not expressively complete (Theorem 9.1.9) whereas FTs with constraints are (Theorem 9.1.10).

Theorem 9.1.9 *COFTs are not expressively complete.*

Short Name	References	GT	NT	GCT	TCL
OFT	Kang et al. (1990)	TREE	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
OFD	Kang et al. (1998)	DAG	$and \cup xor \cup \{opt_1\}$	$\emptyset$	CR
RFD=VBFD	Griss et al. (1998); van Gurp et al. (2001)	DAG	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \vert\}$	CR
EFD	Riebisch et al. (2002); Riebisch (2003)	DAG	$card \cup \{opt_1\}$	$\{\Rightarrow, \vert\}$	CR
GPFT	Eisenecker and Czarnecki (2000)	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\emptyset$	CR
PFT	Eriksson et al. (2005)	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\{\Rightarrow, \vert\}$	$\emptyset$
VFD	Bontemps et al. (2004); Schobbens et al. (2007)	DAG	$card$	$\emptyset$	$\emptyset$
CFD(OP)	-	DAG	OP	$\emptyset$	$\emptyset$
COFD	-	DAG	$and \cup xor \cup \{opt_1\}$	$\emptyset$	$\emptyset$
CRFD	-	DAG	$and \cup xor \cup or \cup \{opt_1\}$	$\emptyset$	$\emptyset$
CFT(OP)	-	TREE	OP	$\emptyset$	$\emptyset$
COFT	-	TREE	$and \cup xor \cup \{opt_1\}$	$\emptyset$	$\emptyset$
CRFT	-	TREE	$and \cup xor \cup or \cup \{opt_1\}$	$\emptyset$	$\emptyset$

Table 9.11: Extended Family of FD languages

Proof We will prove that COFTs cannot express disjunction, i.e. the product line $\{\{A\}, \{B\}, \{A, B\}\}$. COFTs admit nodes without sons, namely the *xor*-node without sons xor_0, where $\mathcal{M}_{FFD}[\![xor_0]\!] = \{\}$, the *and*-node without sons and_0, where $\mathcal{M}_{FFD}[\![and_0]\!] = \{\{\}\}$. These two product lines have no primitive features. Let r be the root, s_i its sons. Any primitive feature can only occur under one s_i, since it is a tree. In this proof we reduce any COFT to a COFT with only two primitive features (say A,B) and enumerate which product lines can be expressed and which cannot (see Figure 9.7). This reduction follows several steps:

1. Translate compound primitive nodes as illustrated in Table 9.12. Put all primitive nodes on leaves by translating the compound primitive nodes (f_2) into a new and_2-node (f_5) with two sons: a copy of the original node (f_2') and the primitive node (f_2) as an and_0-node without son;


Table 9.12: Translation: Compound Primitive Nodes

2. Eliminate any s_i that do not contain A or B: s_i is then equivalent to either and_0 when s_i is an *and*-node or xor_0 when s_i is an *xor*-node;

3. Apply simplification rules (only valid for FTs). In these simplification rules our convention is to decorate optional node n with $\mathring{}\,(\mathring{n})$:

$$xor_k(s_1, \ldots, s_{i-1}, xor_0, s_{i+1}, \ldots, s_k) \;=\; xor_{k-1}(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k) \qquad (9.1)$$

$$and_k(s_1, \ldots, s_{i-1}, xor_0, s_{i+1}, \ldots, s_k) \;=\; xor_0 \qquad (9.2)$$

$$xor_k(s_1, \ldots, s_{i-1}, and_0, s_{i+1}, \ldots, s_k) \;=\; (xor_{k-1}(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k))^{\circ} \qquad (9.3)$$

$$and_k(s_1, \ldots, s_{i-1}, and_0, s_{i+1}, \ldots, s_k) \;=\; and_{k-1}(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k) \qquad (9.4)$$

$$and_1(s_i) \;=\; s_i \qquad (9.5)$$

$$xor_1(s_i) \;=\; s_i \qquad (9.6)$$

4. If the two primitive features A, B occur in the same s_i, we can use the last two rules to replace the root by s_i;

5. We end with either:

   (a) Rule 9.2: xor_0 (a.k.a f the FALSE node in Figure 9.7);

   (b) Rule 9.3: the root with a single optional son, itself having two sons s_1, s_2. In Figure 9.7, we simplify the illustration by omitting the root and decorating the optional node by a hollow circle;

   (c) Rule 9.4: and_0 (a.k.a v the empty node in Figure 9.7);

   (d) Rule 9.5 and 9.6: the root is an *and*-node or a *xor*-node with two sons s_1, s_2, where A occurs in s_1 and B in s_2.

Therefore there are only four product lines that s_1 can express, because it uses only one primitive A: $and_0, xor_0, A, \mathring{A}$. The first two have already been eliminated. This leaves two product lines for s_1 $(A, \mathring{A})$, and the same for s_2 $(B, \mathring{B})$. There are thus 16 possibilities for this case: 2 (has the root a single optional son, or two sons?) times 2 (is the first node with two sons a *xor*- or an *and*-node?) times 2 (is s_1 equivalent to A or $\mathring{A}$?) times 2 (is s_2 equivalent to B or $\mathring{B}$?). There is also and_0 and xor_0. Thus we have 18 simplified trees containing A, B. There are 16 possible product lines to express (Figure 9.7). Unfortunately, several of them express the same PL: $A|B = \{\{\}, \{A\}, \{B\}\}$, however $A \vee B = \{\{A\}, \{B\}, \{A, B\}\}$ is still missing, see Figure 9.7.

$$\square$$

Theorem 9.1.10 *OFTs are expressively complete.*

Proof Let us now consider OFT rather than COFT and therefore add *requires* and *excludes* constraints. Every product line can be expressed with a tree and such constraints. Indeed, similarly to FDs (Definition 9.1.5), a normal form exist for FTs (Definition 9.1.12). The main difference is that a product is not represented with an and_n-nodes and its n sons (its features) but by and_0-nodes with a requires constraint for each of its features. In addition, each primitive feature is linked to the root with an *opt*-node.
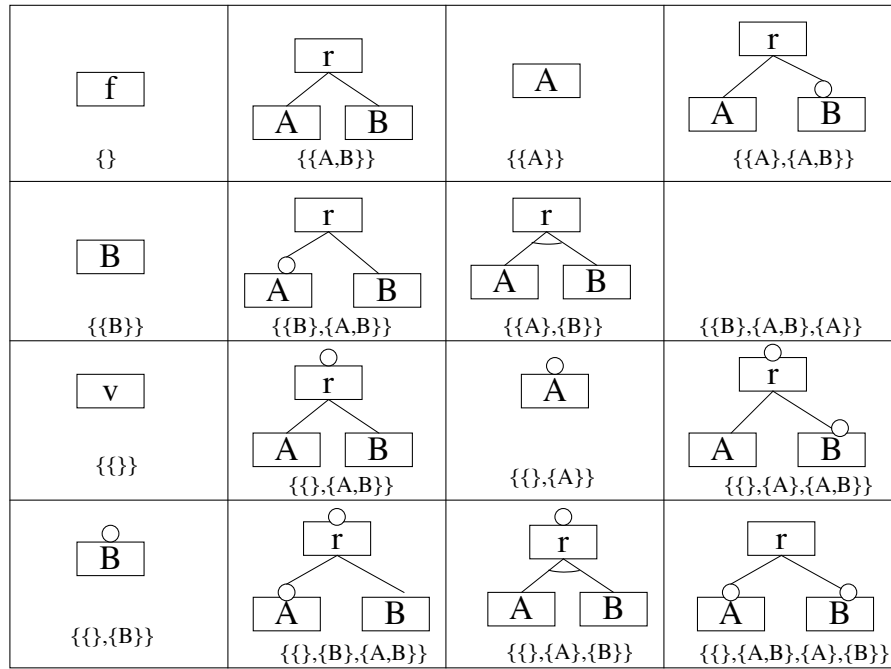
Figure 9.7: Product lines on two primitive features and their COFT

Definition 9.1.12 (Normal Form for a FT) *A FT is in a* Normal Form *when*

- *it is a Tree;*

- *its root is an and_{m+1}-node where m is the number of primitive nodes. The root is linked to m opt-sons (f_i?) and one xor_n-son (PP) that corresponds to the set of products where n is the number of products;*

- *each primitive node (f_i) is a son of its respective opt-son (f_i?) and is represented by an and_0-node without son;*

- *each product (p_i) is a son of PP and is represented with an and_0-node without son. The product with no feature is represented by a xor_0 without son;*

- *each product is related to the primitive features that compose it with "requires" constraints (e.g., p_1 requires f_1 in Figure 9.8). Conversely, each product is related to the primitive features that are not one of its constituent with an excludes constraint (e.g., p_1 excludes f_2 in Figure 9.8);*

$\square$

$\lambda(f1?) = opt_1$

$\lambda(f2?) = opt_1$

$\lambda(r) = and_3$

$\lambda(PP) = xor_3$

$\lambda(fi) = and_0$

r

f1?   f2?   PP

f1   f2   p1   p2   p3   p4

| p1 **requires** f1 | p3 **requires** f2 |
p1 **requires** f2	p3 **excludes** f1
p2 **requires** f1	p4 **excludes** f1
p2 **excludes** f2	p4 **excludes** f2

$\lambda(pi) = and_0$
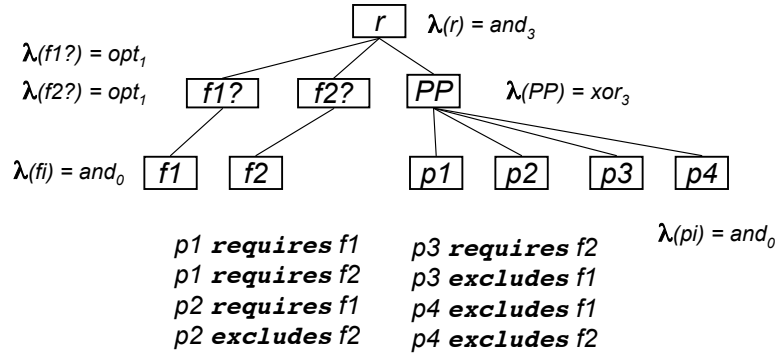
Figure 9.8: FT Normal Form: FT of product line $\{\{\}, \{f_1\}, \{f_2\}, \{f_1, f_2\}\}$

Several proposals have attempted to improve expressiveness of FT languages. However, most FTs are already expressively complete and the new propositions do not improve expressiveness:

- **Add** *requires* **and** *excludes* **constraints**. This proposal allows disjunction and leads to expressive completeness (Theorem 9.1.10). Nevertheless, the use of such constraints is pragmatically ambiguous and reduces comprehension. An illustration is given in Figure 9.8 where the "requires" constraints are used as decomposition relationships. Although its syntax and semantics are both unambiguous, a *requires* constraint may map to two different concepts in the "real world": the concept of *dependence* between one feature and another, and the concept of feature *decomposition*. Without feature sharing, constraints are misused to allow expressive completeness. In addition, they are pragmatically ambiguous in the FT Normal Form (Figure 9.9). Indeed, we use *requires* to indicate which features are included in which products that are themselves non-primitive features. However, the semantics of the *decomposition* and *requires* relationships remain clearly different. For trees, f_1 requires f_2 means that $f_1 \Rightarrow f_2$ while f_1 is decomposed into f_2 means that $f_1 \Leftrightarrow f_2$ since f_2 possesses only one parent (f_1).

- **Add** *or*-**nodes** (Griss et al., 1998). This new type of nodes allows disjunction, but remains expressively incomplete (see theorem 9.1.11). In (Griss et al., 1998), two extensions have been proposed: add *or*-nodes and consider FD as a single-rooted DAG rather than a tree. We will see in the next section that the second extension alone guarantees expressive completeness. On the other hand, adding *or*-nodes only does not give expressive completeness:

Theorem 9.1.11 *CRFTs (i.e., with and- xor- or- and opt-nodes but no constraints) are not expressively complete.*

Proof CRFTs cannot express *card*[2..2] among 3 features. We note that when using trees, *and*-, *xor*-, *or*-nodes are associative. So, without loss of generality, we can assume that the first node from the root that has more than one son, actually has two sons (s_1, s_2). For trees,
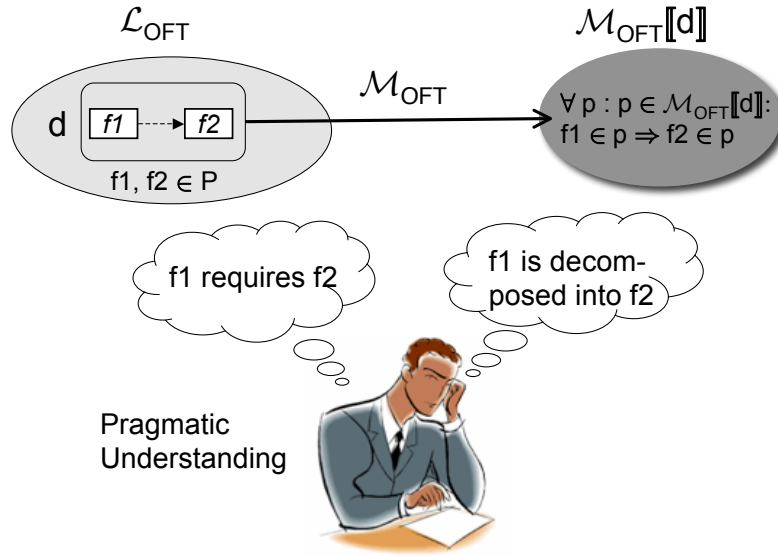
Figure 9.9: Pragmatic Ambiguity in the FT Normal Form

$[\![s_1]\!]$, $[\![s_2]\!]$ have disjoint primitive features. We normalise CRFTs translating each *and*-, *xor*-, *or*-node to respectively *and*$_2$-, *xor*$_2$-, *or*$_2$-node that have an arity of 2 and therefore possess two sons. For this, when the number of sons is higher than 2, normalisation rules are applied:

$$xor_k(s_1, s_2, \ldots, s_k) = xor_2(s_1, xor_{k-1}(s_2, \ldots, s_k)) \tag{9.7}$$

$$and_k(s_1, s_2, \ldots, s_k) = and_2(s_1, and_{k-1}(s_2, \ldots, s_k)) \tag{9.8}$$

$$or_k(s_1, s_2, \ldots, s_k) = or_2(s_1, or_{k-1}(s_2, \ldots, s_k)) \tag{9.9}$$

Now each operator imposes its "shape" on the product line, however none of them allows multiplicities such as $[\![card_3[2..2]]\!]$:

1. for and_2, $PL = \{c_1 \cup c_2 \mid c_1 \in [\![s_1]\!], c_2 \in [\![s_2]\!]\}$;
2. for or_2, $PL = \{c_1, c_2, c_1 \cup c_2 \mid c_1 \in [\![s_1]\!], c_2 \in [\![s_2]\!]\}$;
3. for xor_2, $PL = [\![s_1]\!] \cup [\![s_2]\!]$;

For trees, $[\![s_1]\!]$, $[\![s_2]\!]$ have disjoint primitive features, say without loss of generality $\{A, B, C\}$. $[\![card_3[2..2](s_1, s_2)]\!]$ has none of these shapes.

$\square$

- **Add "group cardinalities"** (operator *card*) to FD languages (Riebisch et al., 2002; Czarnecki et al., 2005b). This construct allows disjunction with *card*[1..*n*] as well as *card*[2..2] among 3 features. However, alone it still does not lead to expressive completeness for trees (see Theorem 9.1.12).

Theorem 9.1.12 *CFTs (i.e., FTs with and-, xor-, or-, card- and opt-nodes but no constraints) are not expressively complete.*

Proof Every type of nodes in a CFT can be translated into a *card*-node following the translation in Table 9.1. However no FTs with only *card*-nodes can express the product line $\{\{\}, \{A\}, \{B\}, \{C\}, \{A, C\}, \{B, C\}\}$. The closest approximation is $card_3[0..2](A, B, C)$, but the product {A,B} should be excluded. The solution is to allow constraints or feature sharing.

$\square$

- **Add nesting to the constraint language** (Batory, 2005). If nesting of constraints is allowed, Sheffer (Sheffer, 1913) has shown that *NAND* (a.k.a *mutex* or *excludes*) alone assures expressive completeness, even on primitive features only. Nesting of constraints means that a constraint could be also expressed on constraints, for instance (*A* requires (*B* excludes *C*)).

  One could also use Boolean (a.k.a propositional) logic or Boolean Circuits (Section 9.2) to easily obtain expressive completeness. For instance, (Batory, 2005) uses trees with *and-*, *xor-*, *opt-* and *or*-nodes, with constraints in Propositional Logic where nesting is allowed. Obviously, this is expressively complete.

- **Allow feature sharing**. The use of DAGs rather than Trees (Griss et al., 1998) leads to expressive completeness. As we will see in the next section Feature DAGs are expressively complete.

9.1.2.2 DAG variants

In this section, we examine what we call "Feature Diagrams" (FDs). Unlike trees, they allow sharing of sub-graphs. This small change provides expressive completeness, i.e. every product line can be expressed by an optionless COFD (see Theorem 9.1.13). To prove this, we use normalisation of FDs to the FFD Normal Form noted $\mathcal{N}_1(pl)$ (Definition 9.1.5).

Theorem 9.1.13 *Every PL pl can be expressed by a optionless COFD, e.g. by its FFD normal form:* $\forall pl \in PL.\exists \mathcal{N}_1(pl) \in optionless\ COFD.pl = [\![\mathcal{N}_1(pl)]\!]$

Proof Every product line can be expressed with an optionless COFD. Indeed, the FFD normal form is expressively complete by definition since each product is a son of the *xor*-root and is represented with an and_n-node with n sons (its features).

$\square$

Another alternative is to represent all the possible combinations of features in a FD and to put all the information concerning the validity of these combinations in constraints. This is what we call a FD Constraint Form (Definition 9.1.13). This requires to use a FD language that allows constraints between nodes (primitive and non-primitive ones):

Definition 9.1.13 (FFD Constraint Form) *A FFD is in a* Constraint Form*, noted* $\mathcal{N}_2(pl)$ *when:*

1. *it is a DAG;*

2. *its root is the only xor-node;*

3. *each possible combination of features (c_i) is directly connected to the root and is represented by an and-node with sons corresponding to its features;*

4. *each primitive feature (f_i) is an and_0-node;*

5. *it does not possess any opt, or nor card-node;*

6. *excludes constraints are used to remove combinations that should not be included in pl. Therefore the root excludes the non valid combinations (see Figure 9.10). For instance, c_0 is excluded by the root r since the combination with no feature (empty product) is not valid.*
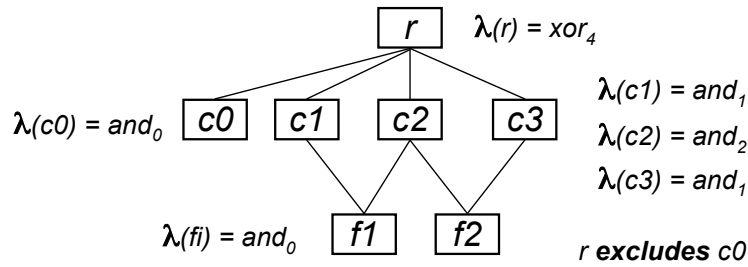
Figure 9.10: FFD Constraint Form: FD of product line $\{\{f_1\}, \{f_2\}, \{f_1, f_2\}\}$

Theorem 9.1.14 *For any set of primitive features P, there is a FD $\mathcal{N}_2(\{P\})$ such that any PL pl on P can be expressed by excludes constraints on $\mathcal{N}_2(\{P\})$.*

Proof The FFD constraint form is expressively complete by definition since each possible product is a son of the *xor*-root and the non-valid ones are discarded by *excludes* constraints coming from the root. However, the size of this constraint form grows exponentially with the number of features.

□

One might ask whether optionless COFD are the smallest expressively complete sub-language. The answer is yes. To prove this, we show that its two operators (*xor*, *and*) cannot be removed (see Theorems 9.1.15 and 9.1.16).

Theorem 9.1.15 *CFD(and) is not expressively complete.*

Proof A product $p \in [\![CFD(\text{and})]\!]$ iff p has exactly one configuration.

□

Theorem 9.1.16 *CFD(xor) is not expressively complete.*

Proof A product $p \in [\![CFD(\text{xor})]\!]$ iff each configuration of p contains exactly one path from the root, and thus one primitive feature at most.

$\square$

However, when *card*-nodes are added, both *and*- and *xor*-node can be removed. The resulting minimal language (VFD) is expressively complete (Theorem 9.1.17).

Theorem 9.1.17 *VFD is expressively complete.*

Proof Every product line can be expressed with a VFD. We know that the FFD normal form is expressively complete by definition. We know that *and*- and *xor*-nodes can be translated into *card*-nodes preserving their semantics. Therefore a new FFD normal form with only *card*-nodes exists (Figure 9.11) and proves the expressive completeness of VFD.

$$\lambda(r) = card_3\,[1..3] \quad \boxed{r}$$

$$\lambda(p1) = card_1\,[1..1] \quad \boxed{p1} \quad \boxed{p2} \quad \boxed{p3} \quad \lambda(p3) = card_1\,[1..1]$$

$$\lambda(p2) = card_2\,[2..2]$$

$$\lambda(fi) = card_0\,[0..0] \quad \boxed{f1} \quad \boxed{f2}$$
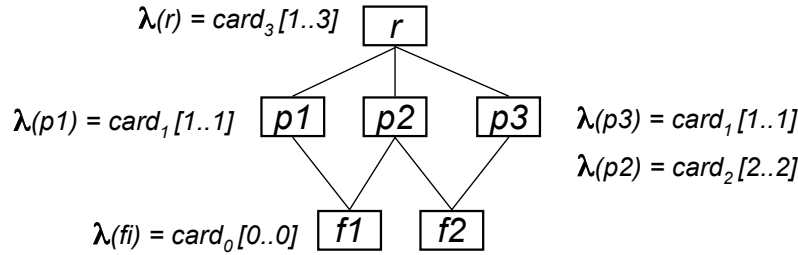
Figure 9.11: FD Card Form: FD of product line $\{\{f_1\}, \{f_2\}, \{f_1, f_2\}\}$

$\square$

Table 9.13 shows that feature languages that allow sharing are expressively complete, while notations based on trees (i.e. forbidding sharing) are not, unless complemented by *opt*-node and constraint language. These constraints are not only used to express dependencies between features but also to compensate feature sharing. Therefore, complementing OFTs with a simple or a rich constraint language leads to expressive completeness but it hinders comprehension when pragmatic ambiguity occurs.

If we ignore the graphical and textual constraints, we have proved that tree languages (OFT (Kang et al., 1990), GPFT (Eisenecker and Czarnecki, 2000), PFT (Eriksson et al., 2005)) are not expressively complete. However, DAG languages (OFD (Kang et al., 1998), EFD (Riebisch et al., 2002; Riebisch, 2003), RFD (Griss et al., 1998)) are expressively complete. Furthermore, we have proved that DAGs with only *card*-nodes are expressively complete. More precisely, our results show that the disjunction of features cannot be expressed in OFT. In (Griss et al., 1998), Griss *et al.* have proposed to solve this problem by (1) adding *or*-nodes, and (2) considering FDs as single-rooted DAGs

TREE/DAG	AND	XOR	OR	OPT	Card	Requires, Excludes	Constraints Nesting	Expressively Complete
TREE	√	√		×	×		×	×
TREE	√	√				×	×	×
TREE	√	√	√			√		√
TREE				√		√		√
TREE							√	√
DAG	×	√		×			×	×
DAG	√	×		×			×	×
DAG	√	√						√
DAG				√				√

Table 9.13: Expressiveness Results

rather than trees. We prove that the second extension alone guarantees expressive completeness while adding *or*-nodes only does not.

When languages are expressively complete, we need a finer yardstick than expressiveness to compare them. The situation is similar for programming languages that are almost all Turing-complete and therefore have the same expressiveness. Two finer criteria are well established: embeddability (a.k.a. naturalness) and succinctness. We study FFD embeddability first.

9.1.3 FFD Embeddability Analysis

Intuitively, a language $\mathcal{L}_1$ is *embeddable* into $\mathcal{L}_2$ iff a semantic- and structure-preserving translation exists from $\mathcal{L}_1$ to $\mathcal{L}_2$ (Felleisen, 1990; Kleene, 1952) (Definition 5.3.3). When a language $\mathcal{L}_1$ is embeddable into a sub-language $\mathcal{L}_2 \subset \mathcal{L}_1$ with the same expressiveness, $\mathcal{L}_1$ is said to be *harmfully redundant* (Definition 5.3.12). Stated otherwise: $\mathcal{L}_1$ is unnecessarily complex because it contains at least one construct that is easily definable using only a strict subset of its own constructs.

In terms of FDs, translations from one language to another and the other way around exist between every pair of expressively complete languages. The question is: Do these translations correspond to an embedding that preserves FD semantics and structure? In other words, each construct should be embeddable. A construct is embeddable into another language when it can be replaced by a fixed diagram. On the one hand, no embedding from FDs to FTs can be defined since the diagram structure cannot be preserved. On the other hand, embeddings exist between FDs that are DAGs. For instance, an embedding exists between COFD and VFD, see Table 9.14 where we use the textual form of the graphs:

- A node bearing an opt_1 operator is translated to a node bearing a $card_1[0..1]$ operator.

- A node bearing a xor_m operator is translated to a node bearing a $card_m[1..1]$ operator.

- A node bearing an and_s operator is translated to a node bearing a $card_s[s..s]$ operator.

This embedding is node-controlled since each *opt*-, *xor*- and *and*-node is treated separately, preserving the structure and the semantics of the original COFD.

Instead of ...	write ...
$opt_1(f)$	$card_1[0..1](f)$
$xor_m(f_1,\ldots,f_m)$	$card_m[1..1](f_1,\ldots,f_m)$
$and_s(f_1,\ldots,f_s)$	$card_s[s..s](f_1,\ldots,f_s)$

Table 9.14: Embedding: COFD into VFD

Graphically, Table. 9.7 shows how CRFD are translated into COFD, applying a translation to *or*-nodes. Once again this embedding preserves semantics while being node-controlled.

9.1.3.1 Redundancy

Embeddings may also appear within the same language. This indicates that some constructs are redundant. As illustrated in Tables 9.15 and 9.16, OFDs have harmfully redundant (Definition 5.3.12) *opt*-nodes, requires and *excludes* constraints. Indeed, an *opt*-node, say $\mathring{n}$ in OFD can be translated into a *xor$_2$*-node n? with two sons: n and the empty node v that can be evaluated to TRUE or FALSE and corresponds to an *and$_0$*-node (i.e. with no son) (Table 9.15). The increase in size can be computed: an optional node is replaced by 2 nodes and 1 edge, leading to an expansion factor of 3.

Instead of ...	write ...

Table 9.15: Redundant *opt*-node in OFD

The *excludes* constraint is also redundant in OFD (see first row in Table 9.16). The first idea would be to translate n_1 mutex n_2 into an *and*-son of r that is a *xor*-node with two sons n_1 and n_2. However, this translation does not preserve the *excludes* semantics that allows the non selection of both features n_1 and n_2. The second idea is to translate a n_1 excludes n_2 into a *xor$_3$*-node ($n_1 \mid n_2$) that is an *and*-son of the root (r) with three sons. Each son corresponds to one of the allowed combinations between n_1 and n_2: $\neg n_2 \wedge n_1$, $\neg n_1 \wedge \neg n_2$, $\neg n_1 \wedge n_2$. However, the semantics, and specifically the justification rule, are still not respected when DAGs are considered. Indeed, the primitive nodes ("n_1", "n_2") that participate in an *excludes* constraint possess new parents ("$\neg n_2 \wedge n_1$", "$\neg n_1 \wedge \neg n_2$", "$\neg n_1 \wedge n_2$") that may harmfully justify them. The solution is to use node negation as illustrated in Table 9.16. The only allowed new parents of the primitive nodes ("n_1","n_2") must be their respective negation ("$\neg n_1$","$\neg n_2$"). If the right branch of the *xor*-node "$n_1 \mid n_2$" is selected, then "n_1" can never be selected and the selection of n_2 is free. Therefore, both cases $\neg n_1 \wedge \neg n_2$ and $\neg n_1 \wedge n_2$ are treated. For the case $\neg n_2 \wedge n_1$, the left branch of the *xor*-node "$n_1 \mid n_2$" is selected. This branch leads to an *and*-node "$\neg n_2 \wedge n_1'$" with two sons "$\neg n_2$" and "n_1'". The node "n_1'" corresponds to the

double negation of n_1 ($\neg\neg n_1$) that avoids the justification rule problem. Indeed, the parent "$\neg n_1$" will never justify "n_1" as, by definition, "t" will be always preferred to "n_1". The increase in size can be computed: an *excludes* constraint composed of 1 node and 2 edges is replaced by 7 nodes and 12 edges, leading to an expansion factor of 7.

Along the same idea, the *requires* constraint is also redundant in OFD (see second row in Table 9.16). The allowed combinations of feature between n_1 and n_2 are: $n_1 \wedge n_2$, $\neg n_1 \wedge \neg n_2$, $\neg n_1 \wedge n_2$. The solution is to use node negation as illustrated in Table 9.16. If the right branch of the *xor*-node "$\neg n_1 \vee n_2$" is selected then "n_1" can never be selected, and the selection of n_2 is free. Therefore, both cases $\neg n_1 \wedge \neg n_2$ and $\neg n_1 \wedge n_2$ are treated. For the case $n_1 \wedge n_2$, the left branch of the *xor*-node "$\neg n_1 \vee n_2$" is selected. This branch leads to an *and*-node "$n_1' \wedge n_2'$" with two sons "n_2'" and "n_1'" that are the the the double negation of n_1 ($\neg\neg n_1$) and n_2 ($\neg\neg n_2$), respectively. The increase in size can be computed: a *requires* constraint composed of 1 node and 2 edges is replaced by 8 nodes and 14 edges, leading to an expansion factor of 8.

Similarly, EFDs have harmfully redundant *opt*-nodes and textual *excludes* and *requires* constraint (see Table 9.17). In addition, *and*-, *or*-, *xor*-nodes from RFD would be also harmfully redundant wrt. *card*-nodes, see Tables 9.18 and 9.19. All these constructs are embeddable into *card*-nodes and therefore into the sub-language of EFD called VFD where only *card*-nodes are allowed.

Riebisch *et al.* have also identified embeddings (Figure 9.12) for *or*-nodes and *xor*-nodes (named alternative) from GPFT to GPFT (Riebisch, 2003).

However, the authors misuse the term "ambiguity" in place of "embedding" as it can be seen in the original caption of Figure 9.12. Let us first recall the definition of ambiguity:

Definition 9.1.14 (Ambiguity) *A diagram (or sentence) is* ambiguous *iff it has two different meanings:* $[\![D]\!] \neq [\![D]\!]$. *A language is ambiguous iff it contains an ambiguous diagram or sentence.*

Ambiguity is obviously impossible with a formal semantics, since $[\![.]\!]$ is a function.

Figure 9.12 actually shows that $[\![d_1]\!] = [\![d_2]\!] = [\![d_3]\!]$ and $[\![d_4]\!] = [\![d_5]\!]$. This is an example of harmless redundancy (Definition 5.3.13) where the three first diagrams are graphically different altough they share the same meaning without using self-embeddable construct.

While adding new constructs may solve the problem of expressive incompleteness, it may also bring ambiguity or redundancy. Riebisch *et al.* have proposed to add multiplicities and optional edges to OFD, stating: *"Such multiplicities cannot be expressed using the previous notations."* (Riebisch, 2003, p.67), thus contradicting Theorem 9.1.13. While the idea is good *per se*, it obviously cannot solve any of the alleged problems:

- If a language is ambiguous, adding constructs will keep it ambiguous.

- If a language is harmlessly redundant, adding constructs will keep it harmlessly redundant.

- If a language is harmfully redundant, adding constructs will keep it harmfully redundant.

The embeddability results for FFD are summarised according to the following theorems.

Instead of ...	write ...	Expansion Factor
n_1 *excludes* n_2		7
n_1 *requires* n_2		8

Table 9.16: Embedding: OFD into COFD

Instead of ...	write ...	Expansion Factor
an opt_1-node	a $card_1[0..1]$	1
n_1 excludes n_2	a $card_2[0..1](n_1, n_2)$, *and*-son of r	2
n_1 requires n_2	see Table 9.19	8

Table 9.17: Embedding: EFD into VFD

| **Instead of . . .**	**write . . .**	**Expansion Factor**
an opt_1-node	a $card_1[0..1]$	1
a xor_s-node	a $card_s[1..1]$	1
an or_s-node	a $card_s[1..*]$	1
an and_s-node	a $card_s[s..s]$	1
n_1 *excludes* n_2	a $card_2[0..1](n_1, n_2)$, *and*-son of r	2
n_1 *requires* n_2	see Table 9.19	8

Table 9.18: Embedding: RFD into EFD and VFD

Instead of . . .	**write . . .**
n_1 *requires* n_2	

Table 9.19: Redundant Requires in EFD
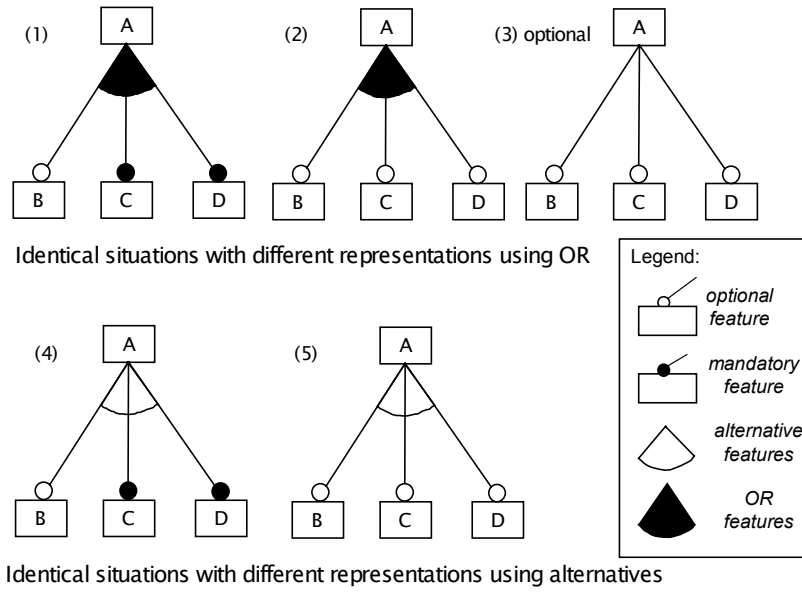
Fig. 2. Examples for ambiguities with OR and alternatives

Figure 9.12: "Ambiguity" example from (Riebisch, 2003)

Theorem 9.1.18 *FTs are embeddable into FDs but not conversely.*

Proof FTs are embeddable into FDs since the translation is simply an identity where each node is translated to itself. The diagram will exactly correspond to the original one. Conversely, FDs are not embeddable to FT since the translation to reach the FT Normal Form (Definition 9.1.12) is not node-controlled.

□

Theorem 9.1.19 *OFD is embeddable into COFD and conversely.*

Proof OFD is embeddable into COFD since the constraints *requires* and *excludes* can be linearly translated into COFD (see Table 9.16). Conversely COFD is embeddable into OFD since COFD is syntactically included in OFD. We say that COFD is a sub-language of OFD.

□

Theorem 9.1.20 *VFD is embeddable into EFD and conversely.*

Proof VFD is embeddable into EFD since VFD is a sub-language of EFD. Conversely, EFD is embeddable into VFD since there exists an embedding from EFD to VFD (Table 9.17).

□

Theorem 9.1.21 *OFD is embeddable into RFD but not conversely.*

Proof OFD is embeddable into RFD since OFD is a sub-language of EFD. Conversely, RFD is not embeddable into OFD since the translation of the *or*-node in RFD to OFD is not linear (Table 9.7). During succinctness analysis (Section 9.1.4) we will see that the expansion factor of the translation is quadratic (Table 9.20).

□

Theorem 9.1.22 *RFD is embeddable into EFD but not conversely.*

Proof RFD is embeddable into EFD since there exists an embedding from RFD to EFD (Table 9.18). Conversely, EFD is not embeddable into RFD since the translation of the *card*-node from EFD to RFD is not linear (Table 9.21). During succinctness analysis, we will see that the expansion factor of the translation is quadratic (Table 9.21).

□

9.1.4 FFD Succinctness Analysis

Intuitively, the *succinctness* (Definition 5.4.1) of a language evaluates the cost of a translation from one language to another, both languages sharing the same expressiveness. This cost is determined by the comparison of the sizes of the FDs. The size of a FD is measured by the number of its nodes and edges. For instance, we have seen that translating a graphical *excludes* constraint into COFD increases the size of the diagram by a factor 7 (Table 9.17) while translating a graphical *excludes* constraint into VFD increases it by a factor 2 (Table 9.17).

Once a complete embedding has been defined between two languages, their succinctness can be deduced. Succinctness results highly depend on embeddings. According to our definition (Definition 5.4.1), a maximal succinctness threshold is fixed on the defined embedding. This upper limit can be reduced if a better embedding is provided. Ideally, we should prove that this embedding is minimal or, in other words, that its expansion factor is minimal.

Theorem 9.1.23 $VFD \leq O(8.EFD)$ *and* $EFD \leq O(VFD)$.

Proof An embedding between EFD and VFD has been defined in Table 9.18. The succinctness can be analysed according to the embedding expansion factor determined in Table 9.17. In the worst case, the embedding from one EFD to a VFD is linear and multiplies the size of the original EFD by a factor 8. Conversely, translating a VFD to an EFD is linear since VFD is a sub-language of EFD (VFD ⊂ EFD) and that each *card*-node in a VFD maps to a *card*-node in an EFD. Therefore, $VFD \leq O(8.EFD)$ and $EFD \leq O(VFD)$.

□

Theorem 9.1.24 $COFD \leq O(8.OFD)$ *and* $OFD \leq O(COFD)$.

Proof An embedding exists between OFD and COFD with an expansion factor of 8 (Table 9.16). Indeed, a *requires* constraint contains 1 node and 2 edges while its translation contains 9 nodes and 14 edges. Conversely, COFD is a sub-language of OFD (COFD $\subset$ OFD). Therefore COFD $\leq O(7.OFD)$ and OFD $\leq O(COFD)$.

$\square$

Theorem 9.1.25 $COFD \leq O(RFD^3)$ *and* $RFD \leq O(COFD)$.

Proof A translation exists between RFD and COFD and this translation is cubic. Indeed, the expansion factor of the translation of an *or*-node is quadratic. If we assume an *or*-node n and its sons ordered as $f_1, \ldots, f_m$, then the node n and its m outgoing edges are translated into $2m + 1$ nodes and $2m + m^2$ edges (Table 9.20). The increase in size is measured as follows:

- In addition to the original node n, m s_i nodes and m f_j? nodes have appeared. Hence, $2m + 1$ nodes have appeared.

- The original edges between the node n and all f_i nodes have been removed.

- The original node n is now related to m s_i nodes. Hence, m edges have appeared.

- Each s_i node is related to 1 f_i node and $m - 1$ f_j? nodes. Hence, m^2 edges have appeared.

- Each f_j? node is related to an f_j node. Hence, m edges have appeared.

Conversely, COFD is a sub-language of RFD. Therefore, COFD $\leq O(RFD^3)$ and RFD $\leq O(COFD)$. Similarly OFD $\leq O(RFD^3)$ and RFD $\leq O(OFD)$ by composition with the previous succinctness results (Theorem 9.1.24).
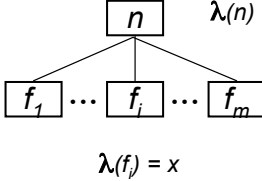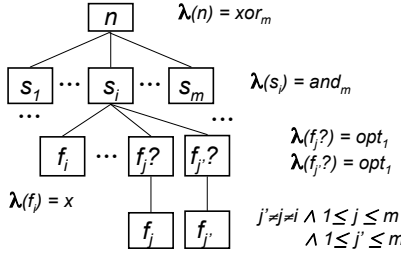
Instead of ...	**write ...**	**Expansion Factor**
		$O(n^2)$

Table 9.20: Succinctness: RFD into COFD

$\square$

Theorem 9.1.26 $COFD \leq O(VFD^3)$.

Proof A translation exists between VFD and COFD and this translation is cubic. Indeed, the expansion factor of the translation for a *card*-node is quadratic. If we assume a *card*-node n with multiplicities $\mu \ldots \nu$ and its sons ordered as $f_1, \ldots, f_m$, then the node n and its m edges are translated into m^2 nodes and m^2 edges (Table 9.21).

Fresh *xor$_2$*-nodes of the form (i, j) are introduced where i ($0 \leq i \leq m$) is the number of sons treated, and j ($0 \leq j \leq i$) is the number of sons in the model among the sons treated. Values of j above ν are collapsed. This new diagram starts from the node (i, j) with $i = 0$ and $j = 0$. Then each (i, j) node is recursively translated following several rules:

- When $i = m$, if $\mu \leq j \leq \nu$ then (i, j) is replaced by the TRUE node t, else (i, j) is replaced by the FALSE node f ($f = xor_0$).

- When $i < m$, if f_i is in the diagram, then two *and*-sons are added to (i, j):

  - The first $(i, j)^-$ is an *and*-node with sons $\neg f_i$ and $(i + 1, j)$. $\neg f_{i+1}$ is a *xor$_2$*-node with two sons, f_i and a TRUE node (t).
  - The second $(i, j)^+$ is an *and*-node with sons f_{i+1} and $(i + 1, j + 1)$;
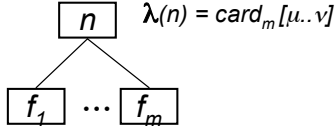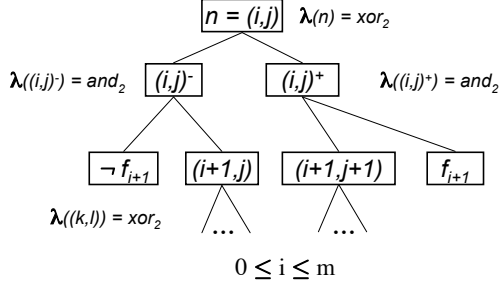
Instead of …	write …	Expansion Factor
		$O(n^2)$

Table 9.21: Succinctness: VFD into COFD

The *card*-node n in VFD and its m edges are exactly translated into $2m + 2 + \sum_{i=0}^{m+1} i + 2. \sum_{i=0}^{m} i$ nodes and $3m + 2 + 6. \sum_{i=0}^{m} i$ edges (Table 9.21). The increase in size is measured as follows:

- The nodes FALSE and TRUE, $m \neg f_{i+1}$ and $m f_{i+1}$ nodes are added. Hence, $2m+2$ nodes are added.

- The total number of nodes (i, j) is k where $k = \sum_{i=0}^{m+1} i$. Indeed, if $m = 4$ then $k = 15$ with 5 nodes for cardinality 4 $((4, 4), (4, 3), (4, 2), (4, 1), (4, 0))$, 4 nodes for cardinality 3, 3 nodes for cardinality 2, 2 nodes for cardinality 1 and 1 node for cardinality 0. Hence, $\sum_{i=0}^{m+1} i$ nodes are added.

- In addition to each (i, j) node where $i < m$, two new nodes are added: $(i, j)^+, (i, j)^-$. Hence, $2. \sum_{i=0}^{m} i$ nodes are added.

- The original edges between the node n and all f_i nodes have been removed.

- Each $\neg f_{i+1}$ node is connected to two edges when they are decomposed. One edge to relate $\neg f_{i+1}$ and f_{i+1}, one edge to relate $\neg f_{i+1}$ and the TRUE node t. Hence, $2m$ edges are added.

- One edge is also added to relate the root to the TRUE node t.

- Each (i, j) node where $i = m$ is connected to one edge to relate (i, j) and the TRUE or FALSE node. Hence, $m + 1$ edges are added.

- Each (i, j) node where $i < m$ is connected to two edges. One edge to relate (i, j) and $(i, j)^+$ and one edge to relate (i, j) and $(i, j)^-$. Hence, $2.\sum_{i=0}^{m} i$ edges are added.

- Each $(i, j)^-$ node is connected to two edges. One edge to relate $(i, j)^-$ and $\neg f_{i+1}$ and one edge to relate $(i, j)^-$ and $(i + 1, j)$. Hence, $2.\sum_{i=0}^{m} i$ edges are added.

- Each $(i, j)^+$ node is connected to two edges. One edge to relate $(i, j)^+$ and f_{i+1} and one edge to relate $(i, j)^+$ and $(i + 1, j + 1)$. Hence, $2.\sum_{i=0}^{m} i$ edges are added.

The illustration provided in Table 9.22 shows how a $card_2[1..2]$ is translated into COFD preserving its semantics following the above procedure with $m = 2$, $\mu=1$ and $\nu=2$. For simplification, negative nodes ($\neg f_i$) are not detailed in Table 9.22 and some shared nodes (e.g. $\neg f_2$) are duplicated to avoid cluttering the figure. In the end, both diagrams share the same semantics: $\{\{f_1\}, \{f_2\}, \{f_1, f_2\}\}$. Accordingly, if $\mu=0$ and $\nu=1$, the diagram is similar except for nodes $(2, 2)$ $(2, 0)$ that are respectively translated into the FALSE node f and the TRUE node t, leading to the semantics: $\{\{f_1\}, \{f_2\}, \{\}\}$. Concerning the expansion factor, the original $card$-node contains 3 nodes and 2 edges. The FD resulting from the translation contains 18 non-duplicated nodes and 26 edges. Five edges are hidden in Table 9.22: two for each negative node and one that relates the TRUE node t to the root.

$\square$

Theorem 9.1.27 *VFD $\leq O(8.RFD)$ and RFD $\leq O(VFD^3)$.*

Proof An embedding between RFD and VFD has been defined in Table 9.18. The succinctness can be analysed according to the embedding expansion factor determined in Table 9.18. In the worst case the embedding from one RFD to a VFD is linear and multiplies the size of the original RFD by a factor 8. Conversely translating a VFD to an RFD is cubic since the expansion factor of translation for a $card$-node is quadratic. The situation is similar as the one for OFD in Table 9.21.

$\square$

The results summarised in Figure 9.13 compose. Further, the inclusions of Theorem 8.3.1 can be used, as well as the linear translations between a textual and the corresponding graphical variant. Thus, at this point, we have essentially three classes of succinctness:

1. (C)OFD is the reference class of succinctness for FDs.

2. RFD is cubically-as succinct as (C)OFD, due to the use of *or*-nodes.

3. EFD is cubically-as succinct as RFD, due to the use of *card*-nodes.

Instead of …	write …
n $\lambda(n) = card_2\,[1..2]$ f_1 f_2 $\lambda(f_1) = x$ $\lambda(f_2) = y$	(0,0) $\lambda((i,j)) = xor_2$ $\lambda((i,j)^-) = and_2$ (0,0)⁻ (0,0)⁺ $\lambda((i,j)^+) = and_2$ ¬f_1 (1,0) (1,1) f_1 $\lambda(f_1) = x$ (1,0)⁻ (1,0)⁺ (1,1)⁻ (1,1)⁺ ¬f_2 (2,0) f_2 (2,1) ¬f_2 f_2 (2,2) f t $\lambda(f_2) = y$ t

<div align="center">Table 9.22: Translation: $card_2[1..2]$ into COFD</div>
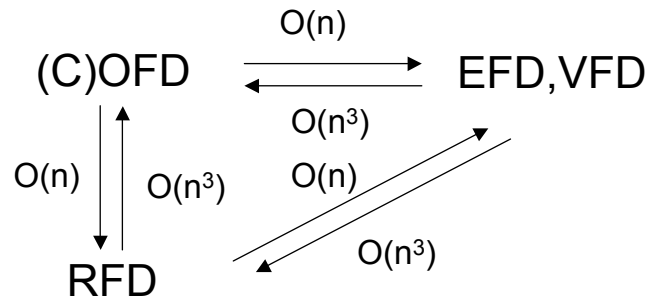

<div align="center">Figure 9.13: FFD Succinctness results</div>

9.2 Boolean Circuits (BC) Analysis

BCs are a well-known formal language defined back in 1937 in (Shannon, 1937, 1938; Vollmer, 1999). BCs represent combinatorial Boolean electronic Circuits. The BC illustrated in Figure 9.14 is composed of *OR* and *AND* gates and has three possible inputs A, B, C. This circuit corresponds to a BF obtained by composing BF at the output of each gate until we reach the final gate, that is Q in our example. The resulting BF of this circuit is $Q = OR(AND(A, B), AND(AND(B, C), OR(B, C)))$ or using Boolean addition and multiplication $Q = AB + BC(B + C)$. This formula can be easily simplified to $Q = B(A + C)$. BCs are exponentially-as succinct as BFs since BCs allow sharing of their inputs.
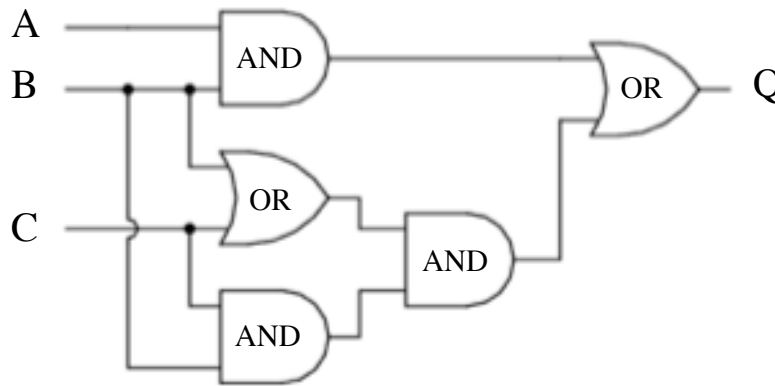

Figure 9.14: Boolean Circuit example

Syntactically, BCs are finite acyclic graphs with nodes called "gates" and edges called "wires". Each gate bears a Boolean operator ($AND, OR, NOT, NAND$). Semantically, a gate computes the value of its operator op applied to the Boolean values e_i carried by its input links: $op_k(e_1, \ldots, e_k)$. This output value is copied to all its output links. For instance, the concrete syntax of a NAND gate is illustrated in the left part of Table 9.23. Its semantics is provided by the BF $x = m \mid n$, see the right part of Table 9.23. As we will see further (Section 9.2.4), It has already been proved by (Sheffer, 1913) that BCs with only NAND gates are expressively complete. Hence all the other gates can be expressed with NAND gates.

Since we have a first understanding on BC, we can now compare it to FFD according to the method proposed in Chapter 7. The first step of the method consists in investigating BC's definition. We first need to determine whether BC's definition is formal and whether it follows the principles described in Chapter 4. Although, BC is formally defined in (Shannon, 1937, 1938; Vollmer, 1999), they require minor adaptations to fit what is advocated in Chapter 4. Hence, we redefine it in Section 9.2.1. Then, BC and FFD semantic domains are related (Section 9.2.2) and their semantics are compared (Section 9.2.3). Finally, we study BC expressiveness (Section 9.2.4), embeddability (Section 9.2.5) and succinctness (Section 9.2.6).

Concrete Syntax	Semantics
	m n $x = m \mid n$ T T F T F T F T T F F T

Table 9.23: NAND-gate and truth values

9.2.1 BC Formal Definition

We formally redefine BC according to the principles described in Chapter 4. As illustrated in Figure 9.15, the BC language is defined with a syntactic domain ($\mathcal{L}_{BC}$), a semantic domain ($\mathcal{S}_{BC}$) and a semantic function ($\mathcal{M}_{BC}$) that are defined respectively in Sections 9.2.1.2, 9.2.1.3 and 9.2.1.4. One way to define the semantics of BC is to map their syntactic domain ($\mathcal{L}_{BC}$) to Propositional Logic's syntactic domain ($\mathcal{L}_{Pr}$) and then to simply reuse Propositional Logic semantic domain ($\mathcal{S}_{Pr}$) and function ($\mathcal{M}_{Pr}$). Defining a language by a translation to another language is a common practice. However, this definition is only valid if we map to a language that is itself formally defined. Therefore, we first need to recall in Section 9.2.1.1 the definition of Propositional Logic according to Harel and Rumpe's principles. Then, we provide the translation ($\mathcal{BF} : \mathcal{L}_{BC} \rightarrow \mathcal{L}_{Pr}$) between $\mathcal{L}_{BC}$ and $\mathcal{L}_{Pr}$.

#### 9.2.1.1 Propositional Logic (Pr) Formal Definition

Propositional Logic is a well-known formal language defined back in 1853 by Boole in his seminal Book: "The Laws of Thought" (Boole, 1853). We formally recall it according to Harel and Rumpe's principles. Therefore, the syntactic domain, semantic domain and semantic function of Propositional Logic are defined in the following sections.

9.2.1.1.1 Propositional Logic Syntactic Domain ($\mathcal{L}_{Pr}$)

Definition 9.2.1 (Propositional Logic Syntactic Domain ($\mathcal{L}_{Pr}$)) *The language of Propositional Logic $\mathcal{L}_{Pr}$ is defined as a tuple (V,Val,O,R) where*

- *V is the set of logical variables that are themselves propositional formulae;*

- *Val is the set of values, Val={T, F} where T and F are propositional formulae;*

- *O is the set of allowed logical operators between logical variables (V), O={$\neg, \vee, \wedge, \oplus, \Rightarrow, \Leftrightarrow$};*

- *R is the set of rules to satisfy when using logical operators (O). A propositional formula ϕ must be of the following form:*

$$\phi ::= a|\phi_1 \vee \phi_2|\phi_1 \wedge \phi_2|\phi_1 \oplus \phi_2|\neg\phi_1|\phi_1 \Leftrightarrow \phi_2|\phi_2 \Rightarrow \phi_2$$
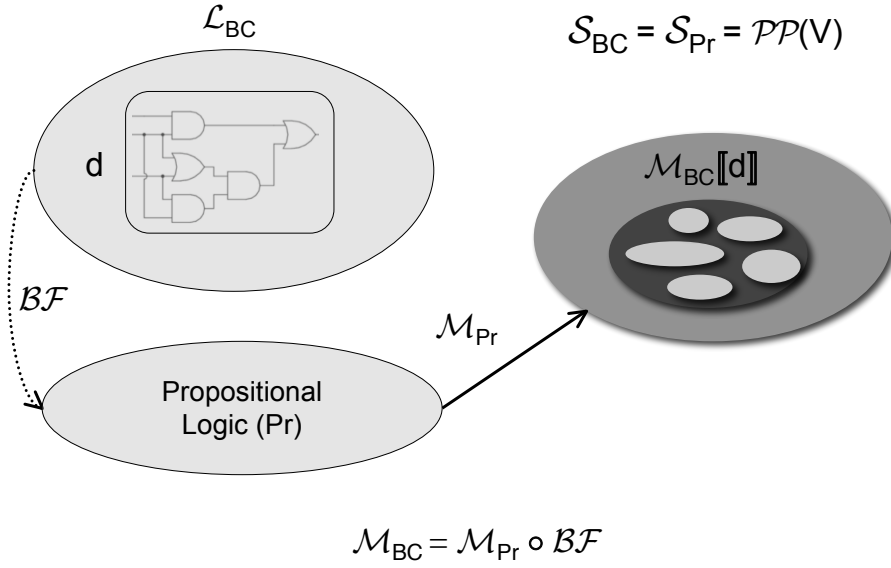
Figure 9.15: BC Semantics

where a $\in V$ and ϕ_1, ϕ_2 are themselves propositional formulae.

9.2.1.1.2   Propositional Logic Semantic Domain ($\mathcal{S}_{Pr}$)

Definition 9.2.2 (Propositional Logic Semantic Domain ($\mathcal{S}_{Pr}$)) *The semantic domain of Pr is mathematically defined as a set of propositional interpretations, themselves sets of logical variables. Hence, $\mathcal{S}_{Pr} = \mathcal{P}(\mathcal{P}(V))$.*

9.2.1.1.3   Propositional Logic Semantic Function ($\mathcal{M}_{Pr}$)

Definition 9.2.3 (Propositional Logic Semantic function ($\mathcal{M}_{Pr}$)) *The semantics of a propositional formula $pf \in \mathcal{L}_{Pr}$ is given by the function $\mathcal{M}_{Pr} : \mathcal{L}_{Pr} \rightarrow \mathcal{P}(\mathcal{P}(V))$. The semantics of a propositional formula is a set of models, where a model is an interpretation that satisfies the formula, and an interpretation (I) is a set of logical variables (that are deemed TRUE). Satisfaction ($\models$) is defined*

by:

$$I \vDash a \quad \equiv \quad a \in I \tag{9.10}$$

$$I \vDash \phi_1 \wedge \phi_2 \quad \equiv \quad I \vDash \phi_1 \text{ and } I \vDash \phi_2 \tag{9.11}$$

$$I \vDash \phi_1 \vee \phi_2 \quad \equiv \quad I \vDash \phi_1 \text{ or } I \vDash \phi_2 \tag{9.12}$$

$$I \vDash \phi_1 \oplus \phi_2 \quad \equiv \quad I \vDash \phi_1 \text{ xor } I \vDash \phi_2 \tag{9.13}$$

$$I \vDash \neg\phi_1 \quad \equiv \quad I \vDash \phi_1 \text{ is false} \tag{9.14}$$

$$I \vDash \phi_1 \Leftrightarrow \phi_2 \quad \equiv \quad I \vDash \phi_1, I \vDash \phi_2 \text{ are equal} \tag{9.15}$$

$$I \vDash \phi_1 \Rightarrow \phi_2 \quad \equiv \quad I \vDash \phi_1 \text{ is false, or } I \vDash \phi_2 \tag{9.16}$$

Satisfaction is defined according to the truth table of each logical operator. The classical truth tables are recalled in Table 9.24.

α	β	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \oplus \beta$	$\alpha \Rightarrow \beta$	$\alpha \Leftrightarrow \beta$
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

Table 9.24: Propositional Logic: Truth Table

9.2.1.2 BC Syntactic Domain ($\mathcal{L}_{BC}$)

Definition 9.2.4 (Boolean Circuit Syntactic Domain ($\mathcal{L}_{BC}$)) *A BC is defined as a tuple:*
(N, E, V, In, Out, G, λ) where

- *N is the set of Nodes where $N = V \cup G$;*

- *V is the set of Boolean Variables;*

- *G is the set of Gates;*

- *$E \subseteq N \times N$ is the multiset of edges that relate Nodes. $(n, n') \in E$ is alternatively noted $n \rightarrow n'$;*

- *$In \subset V$ is the set of input Boolean Variables. $\forall n \in In \cdot (n', n) \notin E$;*

- *$Out \in V$ is the output Boolean Variable. $\forall n \in N \cdot (Out, n') \notin E$;*

- *λ is a labelling function that labels (1) elements of V by a Boolean variable v ($v \in V$) or the constant **TRUE** and (2) elements of G by a Boolean operator (NOT, XOR, OR, AND, NAND);*

- *E forms a DAG. $\nexists n_1, ..., n_k \in N.n_1 \rightarrow ... \rightarrow n_k \rightarrow n_1$;*

- *A Boolean variable appears only once in a BC. $\forall n, n' \in V \cdot \lambda(n) = \lambda(n') \Rightarrow n = n'$;*

- *Only one incoming edge is accepted for non input Boolean variables. $\forall n \in V \wedge \backslash In.\sharp\{(n', n)|(n', n) \in E\} = 1$;*

- *Only one incoming edge is accepted for gates labelled with the NOT operator.* $\forall n \in G \cdot \lambda(n) = NOT \Rightarrow \sharp\{(n', n)|(n', n) \in E\} = 1$.

9.2.1.3 BC Semantic Domain ($\mathcal{S}_{BC}$)

Definition 9.2.5 (BC Semantic Domain ($\mathcal{S}_{BC}$)) *The semantic domain of BC is mathematically defined as the semantic domain of Propositional Logic that is a set of sets of logical variables noted $\mathcal{P}(\mathcal{P}(V))$.*

9.2.1.4 BC Semantic Function ($\mathcal{M}_{BC}$)

Definition 9.2.6 (Boolean Circuit Semantic Function ($\mathcal{M}_{BC}$)) *The semantics of a BC is given by the function $\mathcal{M}_{BC} : \mathcal{L}_{BC} \to \mathcal{P}(\mathcal{P}(V))$. This function is the composition of the translation from BC to Propositional Logic ($\mathcal{BF}$) and the semantic function of Propositional Logic ($\mathcal{M}_{Pr}$), that is, $\mathcal{M}_{BC} = \mathcal{M}_{Pr} \circ \mathcal{BF}$. To translate bc (bc $\in \mathcal{L}_{BC}$) to Propositional Logic, each node n of bc computes a BF according to $\mathcal{BF}$ where, $\mathcal{BF}(bc) = \mathcal{BF}(Out)$ and*

- *if $n \in In$, then $\mathcal{BF}(n) = \lambda(n)$;*

- *if $n \in V \setminus In$, then $\mathcal{BF}(n) = \mathcal{BF}(n')$ where $(n', n) \in E$;*

- *if $n \in G$ and its incoming edges are $(n_1, n) \dots (n_m, n)$, then $\mathcal{BF}(n) = \lambda(n)(\mathcal{BF}(n_1), \dots, \mathcal{BF}(n_m))$.*

Example 9.2.1 *Figure 9.16 illustrates our BC formalisation. This BC, named bc, represents the translation of a NOT gate into its equivalent "NAND" gate. Indeed, Sheffer has shown that NOT p is equivalent to NAND(p, p) (Sheffer, 1913).*
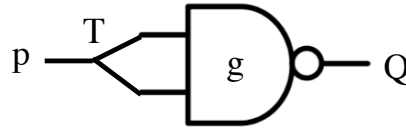
Figure 9.16: BC Semantics Example

According to Definition 9.2.1.3, bc = (N, E, V, In, Out, G, λ) where

- *$N = \{p, T, g, Q\}$;*

- *$E = \{(p, T), (T, g), (T, g), (g, Q)\}^{+}$;*

- *$V = \{p, T, Q\}$;*

- *$In = \{p\}$;*

- *$Out = Q$;*

- $G = \{g\}$;

- $\lambda(p) = p, \lambda(T) = \mathit{TRUE}, \lambda(g) = \mathit{NAND}, \lambda(Q) = Q.$

According to Definition 9.2.1.4, $\mathcal{BF}(bc) = \mathcal{BF}(Q) = \mathcal{BF}(g) = \mathit{NAND}(\mathcal{BF}(T), \mathcal{BF}(T)) = \mathit{NAND}(\mathcal{BF}(p), \mathcal{BF}(p)) = \mathit{NAND}(p, p).$

9.2.2 BC Abstraction Function

In Section 9.2.1, we have recalled and formalised BC's semantics. BCs and FDs are closely related but they do not share the same semantic domain. Hence, we cannot compare them directly.

1. On the one hand BCs are translated to propositional formulae and $\mathcal{S}_{BC}$ consists of set of sets of logical variables ($\mathcal{P}(\mathcal{P}(V))$) where logical variables may correspond to primitive or non-primitive features.

2. On the other hand $\mathcal{S}_{FFD}$ consists of a set of sets of primitive features ($\mathcal{P}(\mathcal{P}(P))$).

However, the two semantic domains are related by an obvious abstraction function ($\mathcal{R}$) (Figure 9.17).
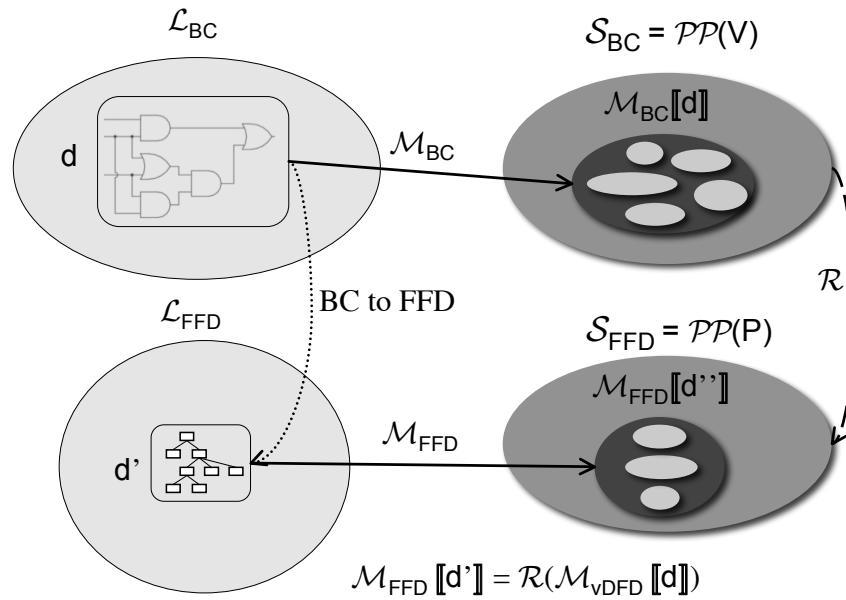
Figure 9.17: Relating BC and FFD

This abstraction function ($\mathcal{R}$) consists in discarding the non-primitive features from the semantic domain. The set of logical variables $\mathcal{P}(V)) \in \mathcal{S}_{Pr}$ representing both primitive and non-primitive features can be reduced to a set of primitive features by simply keeping the primitive ones: $\mathcal{R}(\mathcal{P}(V))) = \{c \cap P | c \in \mathcal{P}(V)\}$.

9.2.3   BC Semantic Equivalence

Once the semantic domains have been related by the abstraction function $\mathcal{R}$ we can check if BCs and FFDs are semantically equivalent (Theorem 9.2.1).

Theorem 9.2.1

$\forall d : d \in \mathcal{L}_{BC} : \mathcal{M}_{FFD}[\![(d')]\!] = \mathcal{R}(\mathcal{M}_{BC}[\![d]\!])$ *where d' is the FFD resulting from the translation of d according to the translation defined in Table 9.26.*

Proof   For simplicity, we consider BC with only NAND gates since we know that they are expressively complete (see Section 9.2.4). Each gate, contained in a BC, can be translated into NAND gates (Sheffer, 1913). Once we have an embedding between NAND gates and FFD, we can extend this result to all the other gates as NAND is expressively complete (see Section 9.2.4). This means that each BC can be translated linearly to a BC containing only NAND gates.

   Since both BC and FFD semantics are node-base semantics, it remains to provide a translation for NAND gates that is an embedding between BC and FFD. We define this translation in Table 9.26 between BC and RFD ($RFD \in FFD$). We know that this translation is node-controlled since the translation is provided for NAND gates. In addition, the translation should preserve the original semantics of BC. Once the NAND gate has been translated into a RFD according to this translation (Table 9.26), we check the equivalence between the BFs given by BC and FFD semantics.

   Table 9.26 illustrates the embedding defined between a NAND gate and its corresponding RFD. This embedding is the result of the composition of two translations.

1. The first one translates a NAND gate into BF in CNF composed of three clauses (C_1, C_2, C_3), see Table 9.25.

2. The second one translates this non existentially quantified formula into its corresponding RFD according to the translation provided in Theorem 9.1.1 but with the set of primitive features (P) limited to its input and output variables (here, $\{ x, m, n \}$). The reader should keep in mind that nodes in FFD (and in Table 9.26) are named with strings. Hence, when a node is called "$\neg x$", this simple node name should not be confused with a BF.
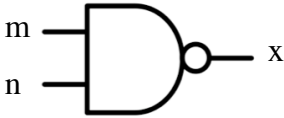
| Instead of ...	write ...
	$[C_1]$ $(\neg x \lor \neg m \lor \neg n) \land$ $[C_2]$ $(x \lor m) \land$ $[C_3]$ $(x \lor n)$

Table 9.25: Translation: NAND gate into Propositional Logic

   On the left of Table 9.27, we present a NAND gate and its corresponding RFD. On the right of the table, we provide their corresponding BFs respectively given by BC and FFD semantics. Several points should be underlined when checking equivalence between these BFs:
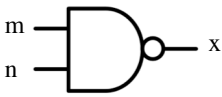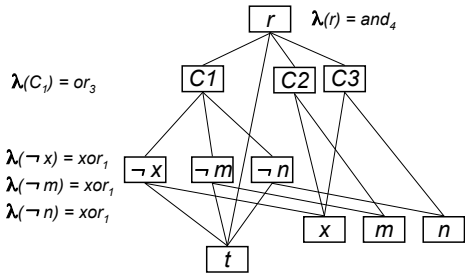
Instead of ...	write ...	Expansion Factor
		6

Table 9.26: Embedding: BC into RFD

1. Non primitive features (r, t, C_1, C_2, C_3) and generated auxiliary symbols $(T_1, \ldots, T_6)$ are existentially quantified in FFD semantics. Therefore, only x, m, n are not quantified as $P = \{x, m, n\}$.

2. Once we know that r is always TRUE and that $\lambda(r) = and$-node we know that C_1, C_2, C_3 and t are also TRUE. Hence, $r \wedge (\neg r \vee C_1) \wedge (\neg r \vee C_2) \wedge (\neg r \vee C_3) \wedge (\neg r \vee t)$ can be eliminated. In addition, the three next lines

   - $(\neg C_1 \vee \text{``} \neg x \text{''} \vee \text{``} \neg m \text{''} \vee \text{``} \neg n \text{''}) \wedge$
   - $(\neg C_2 \vee x \vee m) \wedge$
   - $(\neg C_3 \vee x \vee n)$

   can be reduced to

   - $(\text{``} \neg x \text{''} \vee \text{``} \neg m \text{''} \vee \text{``} \neg n \text{''}) \wedge$
   - $(x \vee m) \wedge$
   - $(x \vee n)$

3. Previously, we have seen that the negation of a node n written "$\neg n$" ("$\neg n$" is the name of the feature not a BF) is translated into a xor_2-node with two sons: the original node n and the TRUE node t. The node t is always evaluated to TRUE and corresponds to an and_0-node (i.e with no son) that is directly related to the root. As t is always TRUE, $\neg n$ is only TRUE when n is FALSE. According to Table 9.2, the translation of a xor_2-node g with two sons x and y gives the following BF where T_1 and T_2 are auxiliary symbols generated by the encoding of a $card_2[1..1]$ node:

   - $(\neg x \vee T_1 \vee \neg g) \wedge$
   - $(\neg y \vee \neg T_1 \vee \neg g) \wedge$
   - $(\neg x \vee T_2 \vee \neg g) \wedge$
   - $(\neg y \vee \neg T_2 \vee \neg g)$

Once one of its son always evaluates to TRUE, let's say y is the TRUE node (hence $y = t$ evaluates to TRUE), then we know that x, T_1 and T_2 evaluate to FALSE. Consequently, x evaluates to FALSE to satisfy the negation of the node x that corresponds to the node g named "$\neg x''$". Hence, "$\neg x$"$\vee$ "$\neg m$"$\vee$ "$\neg n$" is equivalent to $\neg x \vee \neg m \vee \neg n$.
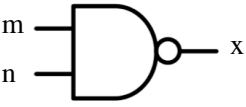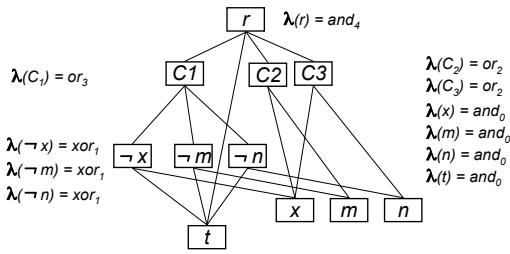
| NAND	BC semantics
	$(\neg x \vee \neg m \vee \neg n) \wedge$ $(x \vee m) \wedge$ $(x \vee n)$
NAND	**FFD semantics**
	$\exists r, t, C_1, C_2, C_3,$ "$\neg x$", "$\neg n$", "$\neg m$", $T_1, T_2, T_3, T_4, T_5, T_6$ $r \wedge (\neg r \vee C_1) \wedge (\neg r \vee C_2) \wedge$ $(\neg r \vee C_3) \wedge (\neg r \vee t) \wedge$ $(\neg (C_1 \vee$ "$\neg x$"$\vee$ "$\neg m$"$\vee$ "$\neg n$"$)) \wedge$ $(\neg C_2 \vee x \vee m) \wedge$ $(\neg C_3 \vee x \vee n) \wedge$ $(\neg$"$\neg x$"$\vee \neg x \vee T_1) \wedge$ $(\neg$ "$\neg x$"$\vee \neg t \vee \neg T_1) \wedge$ $(\neg$ "$\neg x$"$\vee x \vee T_2) \wedge$ $(\neg$"$\neg x$"$\vee t \vee \neg T_2) \wedge$ $(\neg$"$\neg m$"$\vee \neg m \vee T_3) \wedge$ $(\neg$"$\neg m$"$\vee \neg t \vee \neg T_3) \wedge$ $(\neg$"$\neg m$"$\vee m \vee T_4) \wedge$ $(\neg$"$\neg m$"$\vee t \vee \neg T_4) \wedge$ $(\neg$"$\neg n$"$\vee \neg n \vee T_5) \wedge$ $(\neg$"$\neg n$"$\vee \neg t \vee \neg T_5) \wedge$ $(\neg$"$\neg n$"$\vee n \vee T_6) \wedge$ $(\neg$"$\neg n$"$\vee t \vee \neg T_6)$

Table 9.27: NAND: BC vs. FFD Semantics

$\square$

9.2.4   BC Expressiveness Analysis

BCs are expressively complete and self-embeddable. BCs with only NAND gates are expressively complete (Sheffer, 1913). The other gates can be embedded into NAND gates (here noted "|") as follows (Sheffer, 1913):

- *NOT* p is equivalent to $p \mid p$;

- *p AND q* is equivalent to $(p \mid q) \mid (p \mid q)$;

- *p OR q* is equivalent to $(p \mid p) \mid (q \mid q)$;

- *p XOR q* is equivalent to $(p \mid (p \mid q)) \mid (q \mid (p \mid q))$.

9.2.5 BC Embeddability Analysis

In Section 5.3, we proposed a definition of graphical embeddability (Definition 5.3.9) that generalises the definition of embeddability for context-free languages. Given this definition, we need to look at the abstract syntaxes of FFD and BC to study their embeddability.

An embedding between BC and FFD has been already given in Table 9.26. Hence, BCs are embeddable into RFDs but not conversely. Indeed, the non-primitive features contained in FDs imply that intermediate symbols should be added within the corresponding Propositional Logic formula or BC. Therefore, the formula of the circuit should be quantified existentially. In addition, BCs are not embeddable into Propositional Logic since variable sharing is not allowed in propositional formula. Hence, the translation is not linear.

9.2.6 BC Succinctness Analysis

Succinctness (Definition 5.4.1) actually allows comparing the size of the diagram before and after translation. By definition, whenever there is an embedding, there also exists a linear node-controlled translation. As illustrated in Table 9.26, the NAND gate contains 1 nodes and 3 edges while the corresponding RFD contains 8 nodes and 17 edges. The size of the BC with only NAND gates is thus multiplied by a factor 6 n where n is the number of NAND gates in the BC. Hence, RFD $\leq O(6.\text{BC})$. Conversely, BC should be quantified existentially to take into account non-primitive features. Hence, BC $\leq O(2^{RFD})$.

In the end, BC is linearly-as succinct as RFD while RFD is exponentially-as succinct as BC. As illustrated in Figure 9.18, these succinctness results compose with the previous ones.
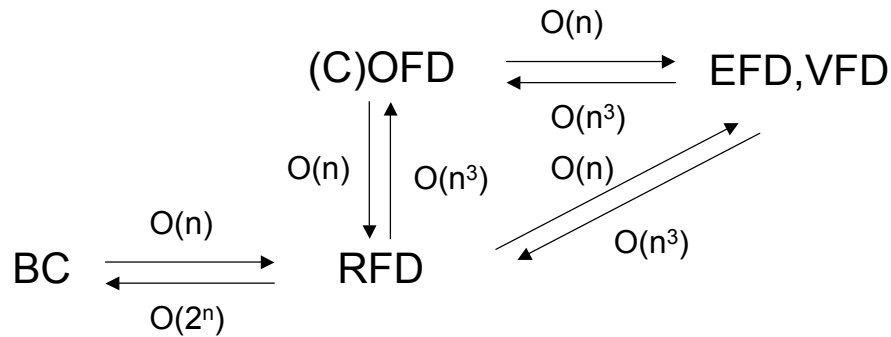
Figure 9.18: BC Succinctness results

9.3   van Deursen *et al.* Language (vDFD) Analysis

9.3.1   vDFD Formal Definition

van Deursen and Klint have formalised in (van Deursen and Klint, 2002) a FD language called vDFD according to our naming convention. The definition of vDFD is based on the Language Specification Formalism ASF+SDF (Brand et al., 2001) and its "meta-environment" tool. ASF and SDF formalisms are intended to describe the syntax and semantics of computer-based formal languages and therefore are in line with Harel and Rumpe's principles (Chapter 4). Syntax Definition Formalism (SDF) allows defining simultaneously both languages concrete and abstract syntaxes. Algebraic Specification Formalism (ASF) allows defining language abstract syntax and semantics based on conditional equations and rewriting rules.

The primary objective of van Deursen and Klint was to reason on FD using a textual representation rather than a graphical one. Further requirements for this representation were (van Deursen and Klint, 2002):

1. to contain all the information contained in the graphical form,

2. to be suited for automatic reasoning.

To satisfy these requirements, the authors define the concept of feature (Definition 9.3.1) and propose a grammar for a *feature description language* (vDFD) (Definition 9.3.3). Finally, they provide rewriting rules to define vDFD semantics. Thus, they define a *feature diagram algebra* with various sets of rewriting rules manipulating vDFD:

1. Normalisation rules ($\mathcal{N}$) to eliminate duplicate features and degenerate cases of the various constructs (Definition 9.3.7);

2. Variability rules to count the number of products allowed in a FD;

3. Expansion rules ($\mathcal{E}$) to expand a normalised feature expression into a disjunctive normal form (Definition 9.3.8);

4. Satisfaction rules ($\mathcal{S}$) to determine which feature expressions in disjunctive normal form satisfy the feature constraints (Definition 9.3.9).

These rewriting rules are used to check the consistency of the representation and to reach a normal form (Definition 9.3.5). $\mathcal{N}$ and $\mathcal{E}$ are used to generate a normal form (syntactic consistency). $\mathcal{S}$ are used to check constraints satisfaction (semantic consistency). Other rules are used to compute variability metrics but they are not relevant in our approach since they do not influence semantics. The application of these rewriting rules follows a specific sequence ($\mathcal{N}, \mathcal{E}, \mathcal{S}$) as illustrated in Figure 9.19. However, we propose an alternative sequence of transformations ($\mathcal{N}', \mathcal{E}', \mathcal{S}'$) on vDFD that corresponds to small corrections suggested for each of them. Once these corrections have been added, the vDFD language definition can be described according to Harel and Rumpe's principles. Therefore, as illustrated in Figure 9.19, the vDFD language is defined with:

1. A syntactic domain ($\mathcal{L}_{vDFD}$) defined as a *feature description language* following the SDF grammar proposed in Definition 9.3.3;

2. A semantic domain ($\mathcal{S}_{vDFD}$) defined within a normal form (Definition 9.3.5);

3. A semantic function ($\mathcal{M}_{vDFD}$) defined as a composition of rewriting rules leading to the normal form. This rewritting rules are recalled in Definitions 9.3.7, 9.3.8 and 9.3.9).

$$\mathcal{M}_{\mathsf{vDFD}} = \mathcal{M} \circ \mathcal{S} \circ \mathcal{E} \circ \mathcal{N}$$
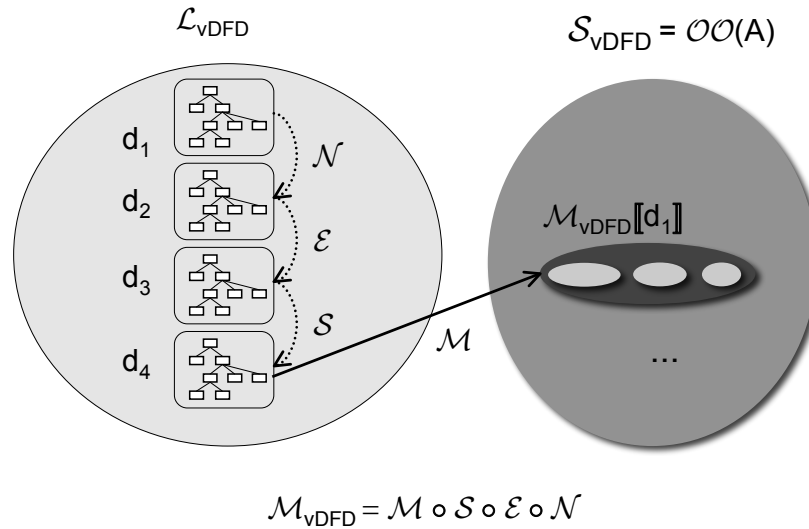
Figure 9.19: vDFD semantics

The next sections present vDFD comparison analysis. The main difference with our preliminary work (Trigaux et al., 2006) is that we follow a clearer method prescribed in Chapter 7. First, we present and discuss vDFD Syntactic Domain ($\mathcal{L}_{vDFD}$) in Section 9.3.1.1. Then we present and discuss vDFD Semantic Domain ($\mathcal{S}_{vDFD}$) and revisit vDFD Semantic Function ($\mathcal{M}_{vDFD}$) respectively in Sections 9.3.1.2 and 9.3.1.3. Finally, we compare this revisited semantics with our own by (1) providing in Section 9.3.2 an abstraction function between $\mathcal{S}_{vDFD}$ and $\mathcal{S}_{FFD}$ and (2) analysing in Section 9.3.3 the semantic equivalence between FFD and vDFD.

9.3.1.1 vDFD Syntactic Domain ($\mathcal{L}_{vDFD}$)

The vDFD Syntactic Domain ($\mathcal{L}_{vDFD}$) is based on the concept of feature (Definition 9.3.1) and on a *feature description language* (Definition 9.3.3), both proposed in (van Deursen and Klint, 2002).

Definition 9.3.1 (Feature definition) *A feature definition (van Deursen and Klint, 2002, p.4) is a feature name followed by ":" and a feature expression (Definition 9.3.2)*

Definition 9.3.2 (Feature expression) *A feature expression (van Deursen and Klint, 2002, p.4) can consist of:*

- *an* atomic feature*;*

- *a* composite feature*: a named feature whose definition appears elsewhere;*

- *an* optional feature*: a feature expression followed by* ?*;*

- mandatory features*: a list of feature expressions enclosed in* all( )*;*

- alternative features*: a list of feature expressions enclosed in* one-of *( );*

- non-exclusive selection of features*: a list of feature expressions enclosed in* more-of( )*;*

- *a* default feature value*:* default = *followed by an atomic feature;*

- *features of the form ..., indicating that a given set is not completely specified.*

Definition 9.3.3 (vDFD Grammar) *A vDFD Grammar (van Deursen and Klint, 2002, p.6) is defined by:*

$[A-Z][a-zA-Z0-9]*$	$\rightarrow$	*FeatureName*
$[a-z][a-zA-Z0-9]*$	$\rightarrow$	*AtomicFeature*
FeatureDefinition*		
Constraint*	$\rightarrow$	*FeatureDiagram*
FeatureName:		
FeatureExpr	$\rightarrow$	*FeatureDefinition*
{ *FeatureExpr* , }+	$\rightarrow$	*FeatureList*
all*(FeatureList)*	$\rightarrow$	*FeatureExpr*
one-of *(FeatureList)*	$\rightarrow$	*FeatureExpr*
more-of *(FeatureList)*	$\rightarrow$	*FeatureExpr*
FeatureName	$\rightarrow$	*FeatureExpr*
AtomicFeature	$\rightarrow$	*FeatureExpr*
FeatureExpr ?	$\rightarrow$	*FeatureExpr*
default = *AtomicFeature*	$\rightarrow$	*FeatureExpr*
...	$\rightarrow$	*AtomicFeature*
DiagramConstraint	$\rightarrow$	*Constraint*
UserConstraint	$\rightarrow$	*Constraint*
AtomicFeature requires		
AtomicFeature	$\rightarrow$	*DiagramConstraint*
AtomicFeature excludes		
AtomicFeature	$\rightarrow$	*DiagramConstraint*
include *AtomicFeature*	$\rightarrow$	*UserConstraint*
exclude *AtomicFeature*	$\rightarrow$	*UserConstraint*

van Deursen and Klint have clearly defined vDFD Syntactic Domain ($\mathcal{L}_{vDFD}$) and the various operations (`all`, `one-of`, `more-of`, ?) to manipulate the allowed expressions. In terms of FFD, we understand $\mathcal{L}_{vDFD}$ as an $\mathcal{L}_{FFD}$(`DAG`, *and* $\cup$ *xor* $\cup$ *or* $\cup \{opt_1\}$, $\emptyset$, CR') where $CR' ::= p_1$(`requires` | `excludes`)p_2 | (`include` | `exclude`)p. Therefore, the translation from $\mathcal{L}_{vDFD}$ operators to $\mathcal{L}_{FFD}$ operators is immediate (Table 9.28).

vDFD operators		FFD operators
`all`	$\rightarrow$	*and*
`one-of`	$\rightarrow$	*xor*
`more-of`	$\rightarrow$	*or*
?	$\rightarrow$	opt_1

Table 9.28: vDFD and FFD operators

However, vDFD is a restricted DAG since the different operators construct a tree, except for the leaves that can be shared by several parents. In addition, specific constraints have been added such as "`include` f" which means that f is common to all products. Therefore $\mathcal{L}_{vDFD} \in \mathcal{L}_{FFD}$ with two additional syntactic rules: one to allow new constraints of the form "`include` f" and "`exclude` f" and a second to restrict feature sharing to the leaves of the tree, see Definition 9.3.4.

Definition 9.3.4 ($\mathcal{L}_{vDFD}$ **as** $\mathcal{L}_{FFD}$) $\mathcal{L}_{vDFD}$ *is* $\mathcal{L}_{FFD}$(`DAG`, *and* $\cup$ *xor* $\cup$ *or* $\cup \{opt_1\}$, $\emptyset$, CR') $(N, P, r, \lambda, DE, CE, \Phi)$ *where:*

- $CR' ::= f_1$(`requires` | `excludes`)f_2 | (`include` | `exclude`)f

- $\forall n1, n2, n3 \in N.n1 \rightarrow n3 \land n2 \rightarrow n3 \land n1 \neq n2 \implies \nexists n4 \in N.n3 \rightarrow n4$

9.3.1.2 vDFD Semantic Domain ($\mathcal{S}_{vDFD}$)

All the reasoning proposed by the authors is based on a disjunctive normal form (Definition 9.3.5) that explicitly lists all possible atomic feature combinations. The purpose is to reduce any vDFD expression to a new expression in a Disjunctive Normal Form that indicates which list of atomic features is a member of the product line. Therefore, this normal form determines $\mathcal{S}_{vDFD}$. $\mathcal{S}_{vDFD}$ is defined as ordered lists of ordered lists of atomic features where atomic features corresponds to leaf features (Definition 9.3.6). This semantic domain suggests that the order of apparition in `all()`-expressions is important. For instance, the disjunctive normal form `one-of(all(`f_2, f_1`), all(`f_1, f_2`))` where $f_1, f_2 \in A$ admits two different products `all(`f_1, f_2`)` and `all(`f_2, f_1`)`. In addition only atomic features are relevant, it means that only the leaves of the DAG are taken into account.

Definition 9.3.5 (Disjunctive Normal Form) *A disjunctive normal form (van Deursen and Klint, 2002, p.9) is a* `one-of` *feature expression with only* `all()`*-expressions as arguments themselves with only* `atomic` *features as arguments. A disjunctive normal form is an expression of the form:* `one-of(all(`$f_{11}, \ldots, f_{1n_1}$`), ..., all(`$f_{m1}, \ldots, f_{mn_m}$`))`

This Disjunctive Normal Form follows the same principles that the FFD Normal Form (Definition 9.1.5). Indeed, each product is represented within an `all()`-expression and all products are included in an `one-of`-expression. However, their purpose is different. The Disjunctive Normal Form is used to determine the semantics of any vDFD while the FFD Normal Form is used as a tool to prove FD expressiveness results.

Definition 9.3.6 (vDFD Semantic Domain ($\mathcal{S}_{vDFD}$)) *The semantic domain of vDFD is mathematically defined as a* product line. *A product line* *is an ordered list of products, i.e., any element of* $PL = O(O(A))$ *where*

- *A is a set of atomic features.*

- *$O(A)$ is an ordered list of A also called a product.*

- *$O(O(A))$ is an ordered list of ordered lists of atomic features (A).*

9.3.1.3 vDFD Semantic Function ($\mathcal{M}_{vDFD}$)

The semantics of any vDFD is provided by a series of rewriting rules. These rewriting rules successively transform the abstract syntax of the vDFD to the disjunctive normal form. Once this form is reached, satisfaction rules (S) are applied to eliminate products that do not satisfy the specified constraints. Finally, the mapping (M) to $\mathcal{S}_{vDFD}$ is immediate. Indeed, the leaves of the normal form corresponds to the atomic features and these atomic features are gathered in a specific order in an `all()`-expression that corresponds to a product. This leads to an ordered list of ordered lists of atomic features, noted $O(O(A))$.

This Disjunctive Normal Form is obtained by applying successively normalisation ($\mathcal{N}$) and expansion rules ($\mathcal{E}$). For instance, the purpose of rules ($\mathcal{N}_2$), ($\mathcal{N}_8$) and ($\mathcal{E}_4$) are:

- ($\mathcal{N}_2$) removes duplicate features;

- ($\mathcal{N}_8$) transforms a `one-of`-expression containing one optional feature into an optional `one-of`-expression;

- ($\mathcal{E}_4$) removes `more-of`-expression `all( )`-expression by translating an `all( )`-expression containing a `more-of`-expression into three expressions: one with the first alternative, one with the first alternative and the remaining `more-of`-expression, and one with the remaining `more-of`-expression.

In Figure 9.20, we translate a diagram $\in \mathcal{L}_{FFD}$ into $\mathcal{L}_{vDFD}$ and show which rules are applied to reach the disjunctive normal form. The first rule to be applied is E_1 that translates an `all( )`-expression containing an optional feature ($f7?$) into two separated cases: one with ($all(f_6, f_7, f_8)$) and one without ($all(f_6, f_8)$). Then the rule E_3 is applied several times to translate an `all( )`-expression containing a `one-of`-expression in two separated cases: one with the first alternative and one with the `one-of`-expression with the first alternative removed. In the end, we have a disjunctive normal form with six `all( )`-expressions that will determine the semantics of this vDFD.

Once the Disjunctive Normal Form is obtained, satisfaction rules (Definition 9.3.9) are applied. If no constraint is imposed (C_s is empty) to the feature expression then ($\mathcal{S}_9$) is called. Otherwise,
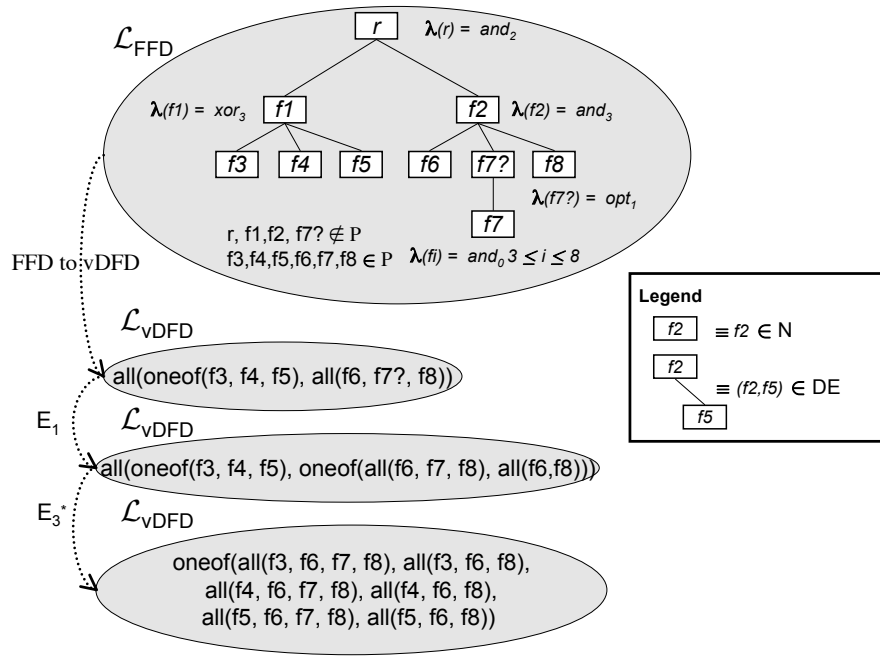
Figure 9.20: vDFD example

binary constraints (`requires`, `excludes`) and unary constraints (`include`, `exclude`) are respectively handled by :

- $(\mathcal{S}_1)$ and $(\mathcal{S}_2)$ for `excludes`;

- $(\mathcal{S}_3)$ and $(\mathcal{S}_4)$ for `requires`;

- $(\mathcal{S}_5)$ and $(\mathcal{S}_6)$ for `include`;

- $(\mathcal{S}_7)$ and $(\mathcal{S}_8)$ for `exclude`.

For instance, the rule $\mathcal{S}_6$ defines that the *sat* function must return FALSE if the constraint *Include f* is applicable and if the atomic feature *f* is not an element of the FeatureExpr *Ft*. When the *sat* function returns FALSE for an `all( )`-expression this expression and therefore its corresponding product are removed.

These rewriting rules are defined in the following. To simplify their definitions the variable naming convention (Table 9.29) proposed in (van Deursen and Klint, 2002) and the following functions (not explicitly defined in (van Deursen and Klint, 2002)) are used :

- *isElement* : *AtomicFeature* × *FeatureExpr* → $\mathcal{B}$ that determines whether the *AtomicFeature* is contained in the *FeatureExpr* or not.

- *sat* : *FeatureExpr* × *Constraints* → $\mathcal{B}$ that determines whether the *FeatureExpr* satisfies the *Constraints* or not.

Meta-Variable	Type
F	FeatureExpr
Fs,Fs',Fs"	{FeatureExpr ","}*
Ft	{FeatureExpr ","}+
f_i	AtomicFeature
C	Constraint
Cs,Cs'	Constraint*

Table 9.29: Variable naming conventions adapted from (van Deursen and Klint, 2002, p.7)

Definition 9.3.7 (Normalization rules) *The set of normalisation rules (van Deursen and Klint, 2002, p.7) is* $\mathcal{N} = \{\mathcal{N}_1, \ldots, \mathcal{N}_{12}\}$:

$(\mathcal{N}_1)$ *Fs, F, Fs', F?, Fs" = Fs, F, Fs', Fs'*
$(\mathcal{N}_2)$ *Fs, F, Fs', F, Fs" = Fs, F, Fs', Fs"*
$(\mathcal{N}_3)$ *F?? = F?*
$(\mathcal{N}_4)$ all*(F) = F*
$(\mathcal{N}_5)$ all*(Fs, all(Ft), Fs') =* all*(Fs, Ft, Fs')*
$(\mathcal{N}_6)$ one-of *(F) = F*
$(\mathcal{N}_7)$ one-of *(Fs,* one-of *(Ft), Fs') =* one-of *(Fs, Ft, Fs')*
$(\mathcal{N}_8)$ one-of *(Fs, F?, Fs') =* one-of *(Fs, F, Fs')?*
$(\mathcal{N}_9)$ more-of *(F) = F*
$(\mathcal{N}_{10})$ more-of *(Fs,* more-of *(Ft), Fs') =* more-of *(Fs, Ft, Fs')*
$(\mathcal{N}_{11})$ more-of *(Fs, F?, Fs') =* more-of *(Fs, F, Fs')?*
$(\mathcal{N}_{12})$ default = A = *f*

Definition 9.3.8 (Expansion rules) *The set of expansion rules (van Deursen and Klint, 2002, p.9) is* $\mathcal{E} = \{\mathcal{E}_1, \ldots, \mathcal{E}_4\}$:

$(\mathcal{E}_1)$ all*(Fs, F?, Ft) =* one-of *(*all*(Fs, F, Ft),* all*(Fs, Ft))*
$(\mathcal{E}_2)$ all*(Ft, F?, Fs) =* one-of *(*all*(Ft, F, Fs),* all*(Ft, Fs))*
$(\mathcal{E}_3)$ all*(Fs,* one-of *(F, Ft), Fs') =* one-of *(*all*(Fs, F, Fs'),* all*(Fs,* one-of *(Ft), Fs'))*
$(\mathcal{E}_4)$ all*(Fs,* more-of *(F, Ft), Fs') =* one-of *(*all*(Fs, F, Fs'),*
       all*(Fs, F,* more-of *(Ft),Fs'),* all*(Fs,* more-of *(Ft), Fs'))*

Definition 9.3.9 (Satisfaction rules) *The set of satisfaction rules (van Deursen and Klint, 2002, p.13) is* $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_9\}$, *where | means OR:*

$(\mathcal{S}_1)$
$$\frac{isElement(f_2, Fs) \mid isElement(f_2, Fs') = true}{sat(\text{all}(Fs, f_1, Fs'), Cs \ f_1 \text{ excludes } f_2 \ Cs') = false}$$

$(\mathcal{S}_2)$
$$\frac{isElement(f_2, Fs) \mid isElement(f_2, Fs') = false}{sat(\text{all}(Fs, f_1, Fs'), Cs \ f_1 \text{ excludes } f_2 \ Cs') = sat(\text{all}(Fs, f_1, Fs'), Cs \ Cs')}$$

$(\mathcal{S}_3)$ $\dfrac{isElement(f_2,\ Fs)\ |\ isElement(f_2,\ Fs')\ =\ false}{sat(\texttt{all}(Fs,\ f_1,\ Fs'),\ Cs\ f_1\ \texttt{requires}\ f_2\ Cs')\ =\ false}$

$(\mathcal{S}_4)$ $\dfrac{isElement(f_2,\ Fs)\ |\ isElement(f_2,\ Fs')\ =\ true}{sat(\texttt{all}(Fs,\ f_1,\ Fs'),\ Cs\ f_1\ \texttt{requires}\ f_2\ Cs')\ =\ sat(\texttt{all}(Fs,\ f_1,\ Fs'),\ Cs\ Cs')}$

$(\mathcal{S}_5)$ $\dfrac{isElement(f,\ Ft)\ =\ true}{sat(\texttt{all}(Ft),\ Cs\ \texttt{include}\ f\ Cs')\ =\ sat(\texttt{all}(Ft),\ Cs\ Cs')}$

$(\mathcal{S}_6)$ $\dfrac{isElement(f,\ Ft)\ =\ false}{sat(\texttt{all}(Ft),\ Cs\ \texttt{include}\ f\ Cs')\ =\ false}$

$(\mathcal{S}_7)$ $\dfrac{isElement(f,\ Ft)\ =\ true}{sat(\texttt{all}(Ft),\ Cs\ \texttt{exclude}\ f\ Cs')\ =\ false}$

$(\mathcal{S}_8)$ $\dfrac{isElement(f,\ Ft)\ =\ false}{sat(\texttt{all}(Ft),\ Cs\ \texttt{exclude}\ f\ Cs')\ =\ sat(\texttt{all}(Ft),\ Cs\ Cs')}$

$(\mathcal{S}_9)$ $sat(\texttt{all}(Ft), Cs) = true$

Definition 9.3.10 (vDFD Semantic function) *The semantics of a $d \in \mathcal{L}_{vDFD}$ is a function $\mathcal{M}_{vDFD} :$ $\mathcal{L}_{vDFD} \to O(O(A))$ where $\mathcal{M}_{vDFD}[\![d]\!] = \mathcal{M}(\mathcal{S}(\mathcal{E}(\mathcal{N}(d))))$.*

Nevertheless, we have discovered undesirable semantics due to missing rules. Consequently, we provide some additional rules and justify them (Figure 9.21):

- The rule in $\mathcal{N}_1$ is not sufficient to avoid feature lists that combine mandatory and optional features. Indeed, a feature list such as $Fs, F?, Fs', F, Fs''$ (where $F?$ and F are switched wrt the rule $\mathcal{N}_1$) would be considered normalised. The set of normalisation rules should be corrected by adding one simple rule (Definition 9.3.11);

- The set of expansion rules is not sufficient to produce a correct disjunctive normal form. Indeed, terms of the form f or $\texttt{one-of}(Ft)$ or $\texttt{all}(Ft)$ are allowed. In order to respect the intentions of the authors we extend $\mathcal{E}$ (Definition 9.3.12);

- The satisfaction function (*sat*) is never explicitly called. Consequently, we propose one rule (Definition 9.3.13) that calls this function and eliminates invalid products (products that do not satisfy the constraints).

Definition 9.3.11 (Normalization rules) *The normalisation rules are a set of rules $\mathcal{N}' = \mathcal{N} \cup \{\mathcal{N}_{13}\}$ where*

$$(\mathcal{N}_{13})\ Fs, F?, Fs', F, Fs'' \ =\ Fs, F, Fs', Fs''$$
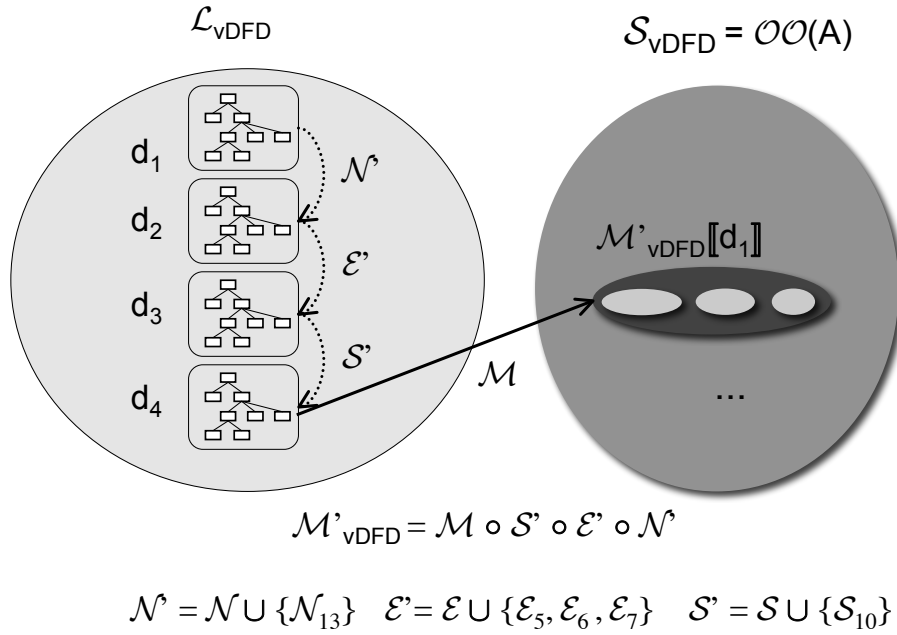
Figure 9.21: vDFD revisited semantics

Definition 9.3.12 (Expansion rules) *The expansion rules are a set of rules* $\mathcal{E}' = \mathcal{E} \cup \{\mathcal{E}_5, \mathcal{E}_6, \mathcal{E}_7\}$
where

$$
\begin{aligned}
(\mathcal{E}_5)\ f &= \texttt{all}(f) \\
(\mathcal{E}_6)\ \texttt{one-of}(F, Fs) &= \texttt{one-of}(\texttt{all}(F), \texttt{one-of}(Fs)) \\
(\mathcal{E}_7)\ \texttt{all}(Ft) &= \texttt{one-of}(\texttt{all}(Ft))
\end{aligned}
$$

Definition 9.3.13 (Satisfaction call rule) *The satisfaction rules are a set of rules* $\mathcal{S}' = \mathcal{S} \cup \{\mathcal{S}_{10}\}$
where

$$
(\mathcal{S}_{10})\ \frac{sat(\texttt{all}(Fs), Cs)}{\texttt{sat}(\texttt{one-of}(Fs', \texttt{all}(Fs), Fs''), Cs) = \texttt{sat}(\texttt{one-of}(Fs', Fs''), Cs)}
$$

Having revisited van Deursen and Klint's rewriting rules we need to redefine the semantics
function (Definition 9.3.14).

Definition 9.3.14 (Revisited vDFD Semantic function) *The revisited semantic of a* $d \in \mathcal{L}_{vDFD}$ *is
a function* $\mathcal{M}'_{vDFD} : \mathcal{L}_{vDFD} \rightarrow O(O(A))$ *where* $\mathcal{M}'_{vDFD}[\![d]\!] = \mathcal{M}(\mathcal{S}'(\mathcal{E}'(\mathcal{N}'(d))))$.

9.3.2 vDFD Abstraction Function

The main difference between vDFD and FFD is that they work with different semantic domains:

1. On the one hand, the semantics of vDFD translates a vDFD expression into another expression in disjunctive normal form that is an ordered list of ordered lists of atomic features $O(O(A))$.

2. On the other hand, the semantic domain of FFD is a set of sets of primitive features $\mathcal{P}(\mathcal{P}(P))$.
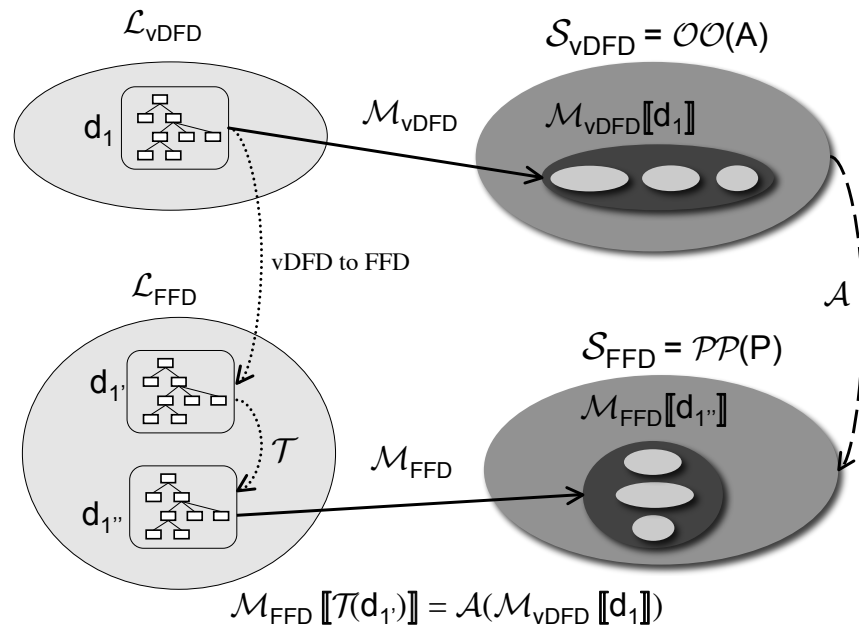
Figure 9.22: Relating vDFD and FFD

Although the semantic domains are different, they can be easily related by an abstraction function (Figure 9.22). Indeed, in (van Deursen and Klint, 2002), atomic features directly correspond to primitive features. The succession of translations push the Atomic features to the leaves of the tree. Hence, the remaining difference is the one between the mathematical structures *list* and *set*. The order of atomic features is important in van Deursen and Klint's semantics. For instance, the textual vDFD expression one-of(all(f_1, f_2), all(f_2, f_1)) contains two products: all(f_1, f_2) and all(f_2, f_1). In FFD's semantics, only one product would be part of PL: $\{f_1, f_2\}$. Consequently, we can define an obvious abstraction function $\mathcal{A}$ (Definition 9.3.15) that simply abstracts away order from van Deursen and Klint's semantic domain and directly maps it to FFD semantic domain. We do not know if this notion of order between features was deliberate or not, but intuitively we consider that two products with the same features should be identical. However, ordering features could

be relevant, for example, when each feature corresponds to one transformation (on code or models) (Ryan and Schobbens, 2004) and these transformations do not produce the same result if they are applied in a different order.

Definition 9.3.15 (Semantic Domain Abstraction $\mathcal{A}$)

$$\mathcal{A} : O(O(A)) \rightarrow \mathcal{P}(\mathcal{P}(P))$$

$$\mathcal{A}(\texttt{all}(f_n, \ldots, f_m), \ldots, \texttt{all}(f_{n'}, \ldots, f_{m'})) = \{\{f_n, \ldots, f_m\}, \ldots, \{f_{n'}, \ldots, f_{m'}\}\}$$

9.3.3   vDFD Semantic Equivalence

van Deursen and Klint's semantics always gives preference to inclusion in terms (not in constraints) and thus behaves like a semantics based on edges rather than on nodes. Therefore, this abstraction function is not sufficient to find an equivalence between FFD's semantics and vDFD's semantics. However, one solution exists to obtain semantic equivalence ($\forall d : d \in \mathcal{L}_{vDFD} : \mathcal{M}_{FFD}[\![\mathcal{T}(d)]\!] = \mathcal{A}(\mathcal{M}_{vDFD}[\![d]\!])$). It consists (1) in translating vDFD into FFD following the embedding in Table 9.30 and then (2) applying a preliminary transformation $\mathcal{T}$ to it (see Figure 9.22). This transformation $\mathcal{T}$ replaces each atomic feature shared by several parents with one *and*-node for each incoming edge, each of these *and*-nodes having only one son that corresponds to an atomic feature. In our concrete notation, the edge-based semantics is obtained by adding one mandatory circle (*and*-node with one son) for each incoming edge of a shared feature. The differences between node-based and edge-based semantics have already been discussed in Section 8.4.1 and illustrated in Figure 8.16.

9.3.4   vDFD Expressiveness Analysis

The usual way to prove that a language $\mathcal{L}_2$ is at least as expressive as $\mathcal{L}_1$ is to provide a translation (Definition 5.2.3) from $\mathcal{L}_1$ to $\mathcal{L}_2$.

Given that and the semantic domain abstraction function $\mathcal{A}$ (Definition 9.3.15), we can consider that the S_{vDFD} is also a product line (PL, see Definition 8.2.2). Thus, every vDFD expresses a PL. Now, we can ask the converse question: can every PL be expressed by a vDFD? Stated otherwise: are vDFD expressively complete?

In vDFD, we have *and*, *xor*, *or* and *opt*-nodes but we do not have DAGs, or at least a restricted form of DAG where only the sharing of leaf nodes is allowed. Studying the expressiveness of vDFD thus requires specific treatment. The operators that manipulate the vDFD expressions must always have at least one operand. Indeed, in vDFD ASF specification (Table 9.3.3) a *FeatureList* contains at least one *FeatureExpr*. Therefore, vDFD expressions without constraints are expressively *incomplete* with respect to PL as the *empty PL* (i.e. the PL containing no product i.e. {}) and the *base PL* (i.e. the PL containing one product in which no feature has been selected i.e. {{}}) cannot be expressed in their disjunctive normal form. If we add vDFD textual constraints, these two products can be expressed. The empty PL can be expressed by: a normal form `one-of` (`all`(A_1)) and a constraint "`exclude` A_1", where A_1 is an atomic feature. A base PL can be expressed by: a normal form `one-of` (`all`(A_1?)) and a constraint "`exclude` A_1" where A_1 is an atomic feature.

The consequence of this result is that we now know that vDFD equipped with constraints (at least `exclude` constraints) is expressively complete. This is interesting because vDFD are supported

by a tool environment and so in theory all FD languages with PL semantics can also be supported by this environment, provided that forward and backward translations between vDFD and the other languages are implemented. We now discuss the practical feasibility of these translations with the two remaining criteria: embeddability and succinctness.

9.3.5 vDFD Embeddability Analysis

In Section 5.3, we proposed a definition of graphical embeddability (Definition 5.3.9) that generalises the definition of embeddability for context-free languages. Given this definition, we need to look at the abstract syntaxes of FFD and vDFD to study their embeddability.

All FT languages are clearly embeddable into vDFD since the difference between node-based and edge-based semantics vanishes for FTs. Conversely vDFD is not embeddable into FT languages since feature sharing is not allowed in FTs.

FD languages are not embeddable into vDFD since sharing of intermediate nodes is not allowed in vDFD. However, if we consider sub-FD languages that restrict the sharing to leaves, we have to apply a linear translation $\mathcal{T}$ on the whole FD to guarantee edge-based semantics. Hence, we have an embedding from restricted RFD to vDFD and conversely an embedding from vDFD to RFD (Table 9.30). For instance the translation of the new constraint "exclude f" is translated in "r excludes f" where r is the root of the FD and similarly for "include f". Therefore, vDFD is embeddable into RFD but not conversely.

Instead of ...	write ...	Expansion Factor
F?	opt_1-node, and_0-node named F and an edge from opt_1-node to and_0-node F	3
one-of($F_1, ..., F_s$)	xor_s-node	1
more-of($F_1, ..., F_s$)	or_s-node	1
all($F_1, ..., F_s$)	and_s-node	1
f_1 excludes f_2	f_1 excludes f_2	1
f_1 requires f_2	f_1 requires f_2	1
exclude f	r excludes f	1
include f	r includes f	1

Table 9.30: Embedding: vDFD into RFD

Embeddings are of practical relevance because they ensure that there exists a transformation from one language to the other that preserves the whole shape of the models. This way, traceability between the two models is greatly facilitated and tool interoperability is made more transparent. Embedding results must however be completed by examining the blow-up caused by the change of notation. This is what is measured by succinctness.

9.3.6   vDFD Succinctness Analysis

Succinctness (Definition 5.4.1) actually allows comparing the size of the diagram before and after translation. By definition, whenever there is an embedding, there also exists a linear node-controlled translation. In this case, a vDFD produced from a tree-shaped FD is identically-as succinct as the tree. A vDFD produced from a restricted DAG is linearly-as succinct as the latter (because intermediate nodes and edges need to be added). Also, the translation from a vDFD to an RFD (Figure 9.30) is linear since there exists an embedding between them.

   In all those cases, the translation engines do not face tractability issues. However, for turning unrestricted FD into vDFD, there is no embedding and all shared cases that vDFD will treat as independent need to be precomputed. This may cause an exponential blow-up. Accordingly, this means that one will be able to apply such transformations only to small diagrams. Therefore, vDFD is linearly-as succinct as RFD (RFD $\leq$ O($vDFD$)) and RFD is exponentially-as succinct as vDFD (vDFD $\leq$ O(2^{RFD})). As illustrated in Figure 9.23, these succinctness results compose with the previous ones.
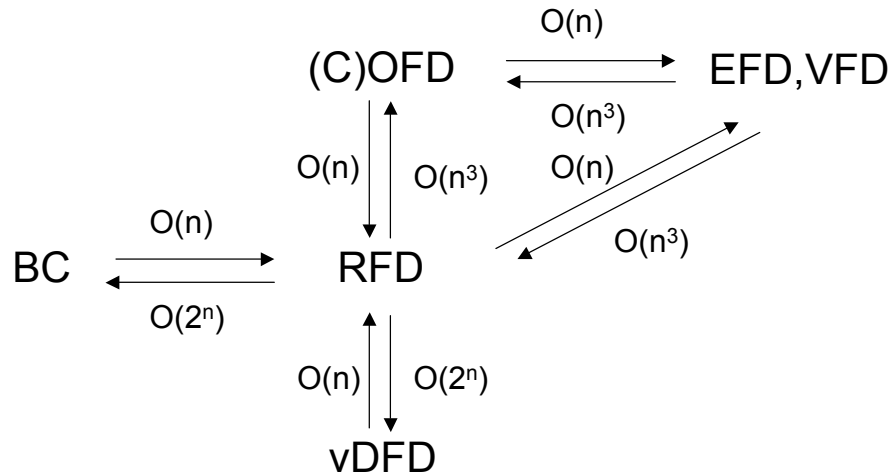
Figure 9.23: vDFD Succinctness results

9.4 Batory Language (BFT) Analysis

9.4.1 BFT Formal Definition

Batory has formalised in (Batory, 2005) a FT language called BFT according to our naming convention. In order to support FD analysis and debugging, the author has proposed to connect FDs, grammars and propositional formulae. As illustrated in Figure 9.24, BFT language is defined by two translations mapping BFT to languages for which a formal semantics is defined or already well-known. The first translation connect BFTs to iterative tree grammars (ITGs) ($\mathcal{T}_1$) and the second to connect ITGs to propositional formulae ($\mathcal{T}_2$). However, this definition does not clearly fit Harel and Rumpe's principles (Chapter 4). Indeed, no syntactic domain has been formally defined. In addition, this definition contains two different semantic domains and two different semantic functions corresponding to the different translations. Therefore, as illustrated in Figure 9.24, two BFT semantics are defined with:

1. A syntactic domain ($\mathcal{L}_{BFT}$) defined as a member of $\mathcal{L}_{FFD}$;

2. A semantic domain ($\mathcal{S}_{ITG}$) defined as a set of String $\mathcal{P}(P^*)$ and its semantic function ($\mathcal{M}_{BFT1}$) that translates BFT to ITGs ($\mathcal{T}_1$).

3. A semantic domain ($\mathcal{S}_{Pr}$) defined as a set of set of logical variables $\mathcal{PP}(V)$ and its semantic function ($\mathcal{M}_{Pr}$) that translates BFT to ITGs and ITGs to Propositional Logic ($\mathcal{T}_2 \circ \mathcal{T}_1$).

We will study these translations in details, before we compare them with FFD. As we go on, the rest of Figure 9.24 will be explained. Therefore, we propose a formal definition for ITG language in Section 9.4.1.1 according to the informal definition provided by Batory in (Batory, 2005). The definition of Propositional Logic have been already recalled in Section 9.2.1.1 when studying BC.

Once these definitions are provided, we define and discuss BFT Syntactic Domain ($\mathcal{L}_{BFT}$), BFT Semantic Domains ($\mathcal{S}_{ITG}$ and $\mathcal{S}_{Pr}$) and BFT Semantic Functions ($\mathcal{M}_{ITG}$ and $\mathcal{M}_{Pr}$) respectively in Sections 9.4.1.2, 9.4.1.3 and 9.4.1.4. Finally, we compare these semantics with FFD semantics by (1) providing in Section 9.4.2 an abstraction functions between $\mathcal{S}_{ITG}$, $\mathcal{S}_{Pr}$ and $\mathcal{S}_{FFD}$ and (2) analysing in Section 9.4.3 the semantic equivalence between FFD and BFT.

9.4.1.1 Iterative Tree Grammars (ITGs) Formal Definition

Batory uses a new formalism called Iterative Tree Grammars (ITGs). We formally define it according to Harel and Rumpe's principles and to our understanding of the informal description given in (Batory, 2005). Therefore, the syntactic domain, semantic domain and semantic function of ITGs are defined in the following sections.

9.4.1.1.1 ITG Syntactic Domain ($\mathcal{L}_{ITG}$) Iterative Grammars (IGs) are not defined by Batory, but we understood its syntactic domain as follows:

Definition 9.4.1 (Iterative Grammar Syntactic Domain ($\mathcal{L}_{IG}$)) *An iterative grammar G is a tuple* (v, VT, VN, R) *where*
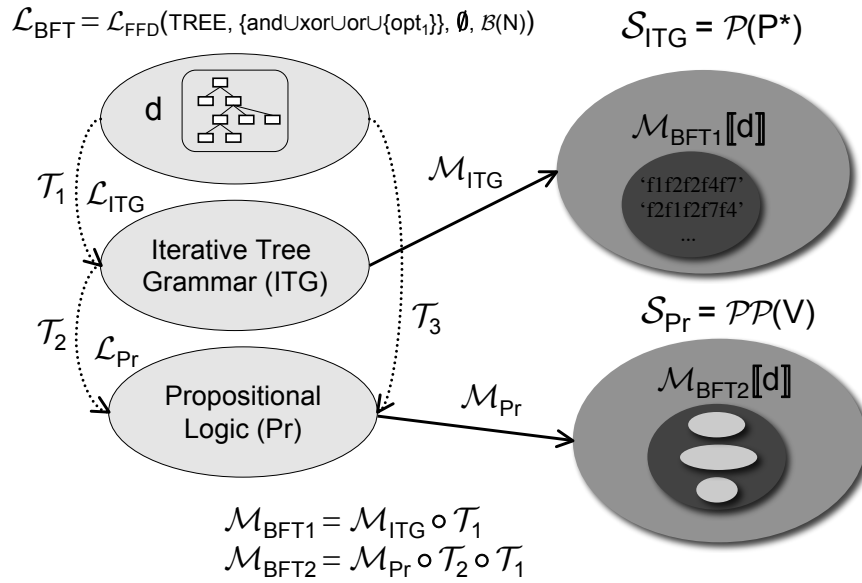
Figure 9.24: BFT semantics

- $v \in VN$ *is the* start symbol *(variously called* root production*,* root*,* start production *in (Batory, 2005)),*

- *VT is the set of* terminals *or* tokens*,*

- *VN is the set of* Non-Terminals*; VN $\cup$ VT is called the* symbols *of G,*

- *R is the set of* rules*, one per non-terminal, that are of the form $h : \pi_1|\ldots|\pi_n$; where h is the non-terminal just mentioned, and a pattern π_i is a named sequence (m) of (possibly optional or repeating) non-terminals n and (possibly optional) terminals $p : \pi_i \underline{::=} (n|\underline{[}n\underline{]}|n^*|n^+|p|\underline{[}p\underline{]})\underline{{}^*} ::$ m. Above, we have underlined the meta-symbols of our meta-grammar, while non-underlined operators actually occur in Batory's grammars.*

Tree Grammars (TGs) are defined by Batory as follows: "A tree grammar requires every token to appear in exactly one pattern, and the name of every production to appear in exactly one pattern. The root production is an exception; it is not referenced in any pattern." (Batory, 2005). Two problems exist within this definition: (1) the same pattern could occur in several rules, and (2) a token could appear several times in the same pattern. Our definition is thus:

Definition 9.4.2 (ITG Syntactic Domain ($\mathcal{L}_{ITG}$)) *An iterative tree grammar (ITG) is an iterative grammar where the start symbol occurs in no right-hand side, and each other symbol occurs exactly once in right-hand sides: $\forall s \in G \setminus \{v\} : \exists!(t : l;) \in R : (l = \pi_0 s \pi_1|\ldots|\pi_k \vee l = \pi_0[s]\pi_1|\ldots|\pi_k \vee l =$*

$\pi_0 s^* \pi_1 | \dots | \pi_k \lor l = \pi_0 s^+ \pi_1 | \dots | \pi_k) \land \forall i \in \{1 \dots k\}, s \notin Symbols(\pi_i),$ *where Symbols gives the symbols appearing in a pattern.*

9.4.1.1.2 ITG Semantic Domain ($\mathcal{S}_{ITG}$)

Definition 9.4.3 (ITG Semantic Domain ($\mathcal{S}_{ITG}$)) *The semantic domain of ITG is mathematically defined as a set of strings, noted $\mathcal{P}(P^*)$.*

9.4.1.1.3 ITG Semantic Function ($\mathcal{M}_{ITG}$)

There are two ways to give this semantics, an operational semantics where derivations eventually produce a string, and a denotational semantics where we associate a set of strings to each symbol, and the semantics of the grammar $[\![G]\!]$, also noted $\mathcal{L}(G)$ is the denotational semantics of its start symbol $[\![v]\!]$. We use the latter in Definition 9.4.4.

Definition 9.4.4 (ITG Semantic function ($\mathcal{M}_{ITG}$)) *The semantics of a $d \in \mathcal{L}_{ITG}$ is a function $\mathcal{M}_{ITG} : \mathcal{L}_{ITG} \to \mathcal{P}(P^*)$ where $\mathcal{M}_{ITG}[\![d]\!] = [\![v]\!]$ where v is its start symbol and*

- *$[\![p]\!] = \{"p"\}$, a terminal just denotes the set of a single string of a single character;*

- *$[\![[s]]\!] = [\![s]\!] \cup \{""\}$, an optional symbol adds the empty string;*

- *$[\![s^*]\!] = \bigcup_{n \in \mathbb{N}} [\![s]\!]^n$, where $[\![s]\!]^0 = \{""\}$ and $[\![s]\!]^n = [\![s]\!].[\![s]\!]^{n-1}$ (where . denotes string concatenation), a symbol iterated by $*$ can be repeated any number of times;*

- *$[\![s^+]\!] = \bigcup_{n > 0} [\![s]\!]^n$, a symbol iterated by $+$ must be repeated at least once;*

- *if $\pi = c_1 \dots c_k$, then $[\![\pi]\!] = [\![c_1]\!].\dots.[\![c_k]\!]$, a pattern is the concatenation of its components;*

- *if $n : \pi_1 | \dots | \pi_k$ then $[\![n]\!] = [\![\pi_1]\!] \cup \dots \cup [\![\pi_k]\!]$, a rule is the union of its patterns.*

While IGs are as expressive as context-free grammars, ITGs are less expressive than regular expressions, see Theorem 9.4.1.

Theorem 9.4.1 *ITGs are less expressive than regular expressions.*

Proof Indeed, any ITG can be translated into a regular expression while any regular expression cannot be translated into ITG:

1. The translation from ITG to regular expressions is obtained by replacing each non-terminal, except the root, by the right side of its defining rule. Since it only occurs once, this transformation is linear. We obtain $v : E$, where E is a regular expression where each token occurs at most once.

2. Any language with a finite repetition, e.g. $\{"aa"\}$, cannot be expressed by ITG. Indeed, a can only occur in one pattern π_a. There, it can only occur as such or optional. All other patterns that do not contain π_a have their semantics empty or the empty string. Thus, π_a can only express a at most once. The non-terminal having π_a in the right hand of its rule can only be repeated 0, 1 or unboundedly many times.

$\square$

9.4.1.2 BFT Syntactic Domain ($\mathcal{L}_{BFT}$)

Batory (Batory, 2005) borrows the FD language named GPFT and already described in Section 6.3.5. However, GPFT's abstract syntax is not clearly defined neither in (Batory, 2005) nor in (Eisenecker and Czarnecki, 2000). We have already clarified it and defined it as FFD (TREE, *and* $\cup$ *xor* $\cup$ *or* $\cup$ $\{opt_1\}$, $\emptyset$, *CR*) (see Table 8.1).

Nevertheless, BFT appears to be slightly different. We understand that BFT's abstract syntax is FFD (TREE, *and* $\cup$ *xor* $\cup$ *or* $\cup$ $\{opt_1\}$, $\emptyset$, $\mathcal{B}(V)$), where $\mathcal{B}$ is the Propositional Logic built by taking V as logical variables and with the restriction that the primitive features must be the leaves of the tree. One point concerning the BFT syntactic domain is unclear in (Batory, 2005). Indeed, the author mentions outside of the main definition that *or*-nodes could be given a cardinality (m, n). However, cardinalities do not occur in BFT concrete syntax as it appears in the example given. If wanted, FFD definition can be easily accommodated by adding *card* or replacing *or* by *card* in *NT*.

9.4.1.3 BFT Semantic Domains ($\mathcal{S}_{BFT}$)

As illustrated in Figure 9.24, each BFT semantic domains correspond to the semantic domain of the language to which BFT is mapped. Thus, the first semantic domain is ITG semantic domain $\mathcal{S}_{ITG}$ (Definition 9.4.3) and the second semantic domain is Pr semantic domain $\mathcal{S}_{Pr}$ (Definition 9.2.2).

9.4.1.4 BFT Semantic Functions ($\mathcal{M}_{BFT1}$ and $\mathcal{M}_{BFT2}$)

As illustrated in Figure 9.24, two different semantic functions exist for BFT. The first one $\mathcal{M}_{BFT1}$ (Definition 9.4.5) is the composition of translation $\mathcal{T}_1$ and ITG semantic function $\mathcal{M}_{ITG}$. The second one $\mathcal{M}_{BFT2}$ (Definition 9.4.6) is the composition of translations $\mathcal{T}_1$, $\mathcal{T}_2$ and Propositional Logic semantic function $\mathcal{M}_{Pr}$.

Definition 9.4.5 (First BFT Semantic function ($\mathcal{M}_{BFT1}$)) *The first semantics of a BFT $d \in \mathcal{L}_{BFT}$ is a function $\mathcal{M}_{BFT1} : \mathcal{L}_{BFT} \rightarrow \mathcal{S}_{ITG}$ where $\mathcal{M}_{BFT1}[\![d]\!] = \mathcal{M}_{ITG} \circ \mathcal{T}_1$.*

Definition 9.4.6 (Second BFT Semantic function ($\mathcal{M}_{BFT2}$)) *The second semantics of a BFT $d \in \mathcal{L}_{BFT}$ is a function $\mathcal{M}_{BFT2} : \mathcal{L}_{BFT} \rightarrow \mathcal{S}_{Pr}$ where $\mathcal{M}_{BFT2}[\![d]\!] = \mathcal{M}_{Pr} \circ \mathcal{T}_2 \circ \mathcal{T}_1$.*

Every translation is defined except $\mathcal{T}_1$ and $\mathcal{T}_2$. These definitions have been provided in (Batory, 2005). Therefore we simply recall and discuss them in the two following sections.

9.4.1.4.1 From BFT to ITG ($\mathcal{T}_1$) The first translation ($\mathcal{T}_1$) defines how BFTs are translated into ITG. We define the translation $\mathcal{T}_1$ sketched in (Batory, 2005), that takes a BFT in input (called there a feature diagram) and produces an ITG in output.

Definition 9.4.7 *For a BFT d, the transformation $\mathcal{T}_1$ gives the grammar $\mathcal{T}_1(d)$ defined by:*

1. *the start production is the root of the BFT;*

2. *the terminals are the primitive features;*

3. *the non-terminals are the internal nodes (compound features), plus non-deterministically cho-sen auxiliary symbols (for or-nodes);*

4. *the rules are:*

   (a) *for an and-node n: a single pattern $n : \pi$, where $\pi = s_1 \ldots s_k$ and where s_k are the sons of n. Remember that hollow circles are also considered as nodes in our abstract syntax;*

   (b) *for a xor-node n: the rule $n : s_1| \ldots |s_k$;*

   (c) *for an optional node n: the rule $n : [s_1]$;*

   (d) *for an or-node n, we choose some fresh non-terminal t and add two rules: $n : t+$ and $t : s_1| \ldots |s_k$.*

In Figure 9.25, we illustrate how a FD $\in \mathcal{L}_{BFT}$ is translated into $\mathcal{L}_{ITG}$ according to translation $\mathcal{T}_1$. The main difference with the example given in (Batory, 2005) is that optional nodes in BFT concrete syntax corresponds to two nodes in FFD abstract syntax (the original node and an opt_1 node). Hence we have one more pattern corresponding to $f7?$ that has no influence.
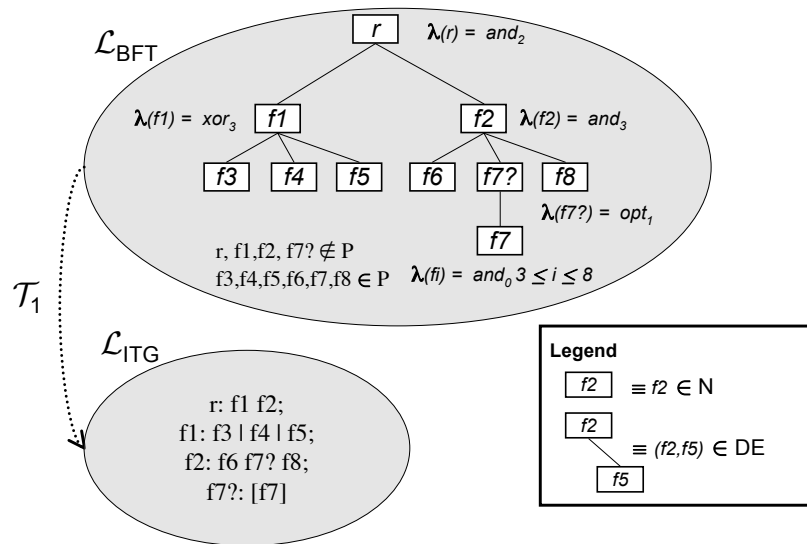
Figure 9.25: $\mathcal{T}_1$: BFT to ITG

We make two observations. First, $\mathcal{T}_1$ only uses a strict subset of ITG: the * is never used, the + can only occur alone in a pattern, etc. Second, $\mathcal{T}_1(or \ldots)$ is not so intuitive: for instance, $\mathcal{T}_1(or(f_1, f_2))$ gives a grammar whose semantics is $\mathcal{M}_{ITG}(\mathcal{T}_1(or(f_1, f_2))) = \{f_1, f_2, f_1f_2, f_2f_1, f_1f_2f_1, f_1f_1f_2, f_2f_1f_2, \ldots\}$. However, if we ignore the order and repetition of the tokens then it looks intuitive again: $\{f_1, f_2, f_1f_2\}$.

9.4.1.4.2 From ITG to Propositional Formula ($\mathcal{T}_2$)   Although $\mathcal{T}_1$ already gives a semantics, another translation is provided in (Batory, 2005). This translation ($\mathcal{T}_2$) transforms the ITG to Propositional Logic $\mathcal{B}(V)$ where V is the set of logical variables. V includes the tokens (primitive features), the non-terminals (the compound features and the auxiliary non-terminals) and the pattern names. Note that the latter two were chosen non-deterministically.

This second transformation, called $\mathcal{T}_2$, operates as follows:

1. for rules of the form $r : \pi_1|\ldots|\pi_k$ that are referenced without repetition (i.e. if they come from a *xor*-node), we add the formula $r \Leftrightarrow \mathsf{choose}_1(\pi_1,\ldots,\pi_k)$. We define the abbreviation $\mathsf{choose}_1(\pi_1,\ldots,\pi_k)$ as $\bigvee_{i\in\{1..k\}} \pi_i \wedge \bigwedge_{j\neq i} \neg\pi_j$. Its expansion factor is thus quadratic in k;

2. for rules of the form $r : \pi_1|\ldots|\pi_k$ that are referenced with repetition (i.e. if they come from an *or*-node), we add the formula $r \Leftrightarrow \pi_1 \vee \ldots \vee \pi_k$;

3. for rules of the form $r : s_1 \ldots s_k$ (i.e. those coming from an *and*-node), we add the formula $(r \Leftrightarrow s_1) \wedge \ldots \wedge (r \Leftrightarrow s_k)$;

4. for rules of the form $r : [s]$ (i.e. those coming from an optional node), we add a rule $s \Rightarrow r$.

Finally, we add a formula stating that the start symbol is TRUE: r. We used "add" above to mean "conjoin": all the formulae produced above are separated by $\wedge$.

In Figure 9.26, we illustrate how an ITG $\in \mathcal{L}_{ITG}$ is translated into $\mathcal{L}_{Pr}$ according to translation $\mathcal{T}_2$. The main difference with the example given in (Batory, 2005) is that Batory writes the formula "$r = true$" to state that the root should be always in the product, while the simple formula r is sufficient since neither "$=$" nor "*true*" are part of Propositional Logic definition.

It seems that the detour through ITG is useless, since the cases defining T_2 could be directly and more simply defined on BFT diagrams. This gives a third transformation T_3 (Figure 9.24) where:

1. for a *xor*-node f, we add the formula $f \Leftrightarrow \mathsf{choose}_1(s_1,\ldots,s_k)$;

2. for an *or*-node f, we add the formula $f \Leftrightarrow s_1 \vee \ldots \vee s_k$;

3. for an *and*-node f, we add the formula $f \Leftrightarrow s_1 \wedge \ldots \wedge f \Leftrightarrow s_k$;

4. for an optional node f, we add a rule $s \Rightarrow f$.

Theorem 9.4.2 *This transformation does the same work:* $\mathcal{T}_3 = \mathcal{T}_2 \circ \mathcal{T}_1$

Conversely, the grammar could be used directly to provide a semantics. However, it finally proved useful to us, since it allows double-checking of the transformations. We will see that by correcting $\mathcal{T}_2$ thanks to $\mathcal{T}_1$, we will discover the semantics probably intended in (Batory, 2005).

No clear semantic definition has been provided in (Batory, 2005). However, our interpretation of BFT's semantics is based on these citations from (Batory, 2005):

- "a feature model (grammar + constraints) *is* a propositional formula."

- "It is possible to show that two FDs are equivalent if their propositional formulae are equivalent."
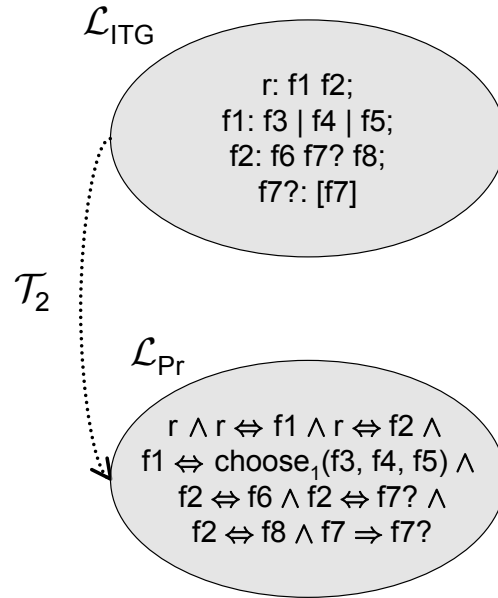
Figure 9.26: $\mathcal{T}_2$: ITG to Pr

For us, it means that the semantics of a FD is the semantics of the formula, i.e. a set of sets of logical variables. However, we face an obstacle: since all auxiliary symbols become logical variables, the equivalence is trivial. In addition, since these auxiliary symbols are generated non-deterministically, the semantics is not a function. This is a really bad property for a semantics. The solution is to eliminate the auxiliary symbols as we have done in FDD definition of product where only primitive features are allowed.

9.4.2 BFT Abstraction Function

In Section 9.4.1.4, we recalled and formalised the two semantics provided by Batory for his feature trees (that we named BFT). Since trees are diagrams (DAG), FFD semantics can also be applied to BFT. Ideally, we should obtain the same semantics. This is clearly impossible here, since the semantic domains are different (Figure 9.24):

1. The first semantics of Batory translates BFT to grammars. Grammars themselves have as semantic domain a set of strings of primitive features ($\mathcal{P}(P^*)$).

2. The second semantics of Batory translates BFT to propositional formulae. These formulae themselves have as semantic domain a set of models, i.e. a set of sets of variables ($\mathcal{P}(\mathcal{P}(V))$), not necessarily primitive features.

3. FFD semantic domain is a set of sets of primitive features ($\mathcal{P}(\mathcal{P}(P))$).

However, the three semantic domains are related by obvious *abstraction functions* (Figure 9.27):
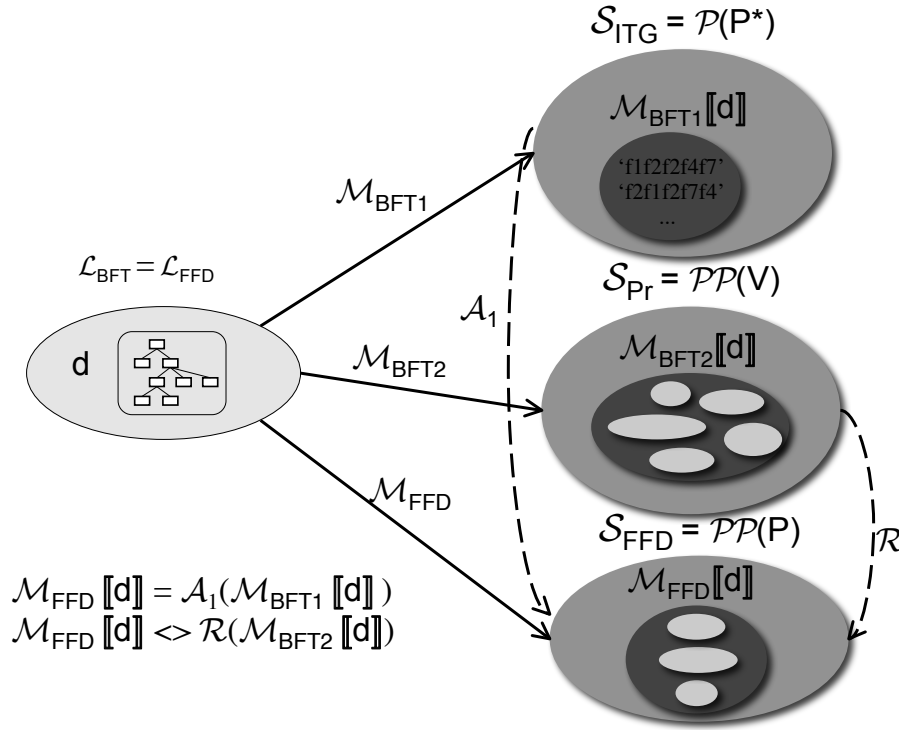
Figure 9.27: Relating BFT and FFD

- $\mathcal{A}_1$: a string can be seen as a set by just forgetting about the order and repetition of the primitive features. In this way, we can abstract a set of strings $\mathcal{L} \in \mathcal{S}_{ITG}$ to a product line (set of sets): $\mathcal{A}_1(L) = \{c \in \mathcal{PP} | \exists s \in \mathcal{L}. \forall a \in P, a \in c \Leftrightarrow a \in s\}$, where $a \in s$ means that a occurs in s: $\exists i \in \mathbb{N}. s_i = a$.

- $\mathcal{R}$: a set of logical variables $\mathcal{P}(V)) \in \mathcal{S}_{Pr}$ representing both primitive and non-primitive features can be reduced to a set of primitive features by just keeping the primitive features: $\mathcal{R}(\mathcal{P}(V)) = \{c \cap P | c \in \mathcal{P}(V))\}$.

One observes that the second "semantics" of Batory is actually not one, for a semantics is required to be a function. Batory's translation procedure non-deterministically chooses auxiliary symbols, and can thus yield different resulting formulae for $\mathcal{M}_{BFT2}$. However, this problem is solved by abstracting this "semantics", since our abstraction drops exactly those assumed faulty symbols.

We can see that FFD semantics (Definition 8.2.2) is exactly what is needed to abstract the two semantics of Batory: this is a further argument for its adequacy.

9.4.3 BFT Semantic Equivalence

The semantic equivalence analysis reveals that only two of the three studied semantics for BFT ($\mathcal{M}_{FFD}, \mathcal{M}_{BFT1}, \mathcal{M}_{BFT2}$) coincide, see Theorem 9.4.3.

Theorem 9.4.3 *For any BFT d, $\mathcal{M}_{FFD}[\![d]\!] = \mathcal{A}_1(\mathcal{M}_{BFT1}[\![d]\!])$*

Proof To prove this equivalence by induction on BFT, we generalise it as follows: A node n of a BFT d is in a model of d iff it occurs in a derivation tree T of $\mathcal{T}_1(d)$. This model $M(T)$ is thus just the set of nodes occurring in the derivation tree. The base case is the root (or concept, or initial non-terminal symbol of the grammar) that is included in any model and in any derivation.

$\Rightarrow$**:** If n is in a model M, by the justification rule its parent f is also in M. By induction hypothesis there is a derivation tree T_f including f. f can be:

1. an *and*-node: mapped to a rule $f : s_1 \ldots s_n$, with some s_i that is either n or $[n]$. Thus we cut the derivation at f and use the rule above, then the rule for optional productions if $\mathring{n} \in M$.

2. a *xor*-node: mapped to a rule $f : P_1|\ldots|P_n$ where some P_i is either n or $[n]$. We select this P_i, then the rule for optional productions if $\mathring{n} \in M$.

3. an *or*-node: mapped to two rules $f : t+$ and $t : P_1|\ldots|P_n$ where t is a new identifier and some P_i is either n or $[n]$. We use the rule for repetition $\#\{P_i|P_i \in M\}$ times, then we use the rule for t for each of these P_i.

$\Leftarrow$**:** Conversely, assume a derivation tree T and compute $M(T)$. It remains to prove that this M (for short) is indeed a model of D. If $n \in T$ then since the derivation leading to n necessarily goes through f, $f \in T$ and by induction hypothesis $f \in M$. Thus the justification rule is respected. f can be:

1. an *and*-node: mapped to a rule $f : s_1 \ldots s_n$, with some s_i that is either n or $[n]$ (that we confuse with $\mathring{n}$). Since n only occurs in this rule (Batory, 2005), we know this rule has been applied and thus $s_1 \ldots s_n \in M$. Thus the semantics of *and* is respected.

2. a *xor*-node: mapped to a rule $f : P_1|\ldots|P_n$ where some P_i is either n or $[n]$. Since $f, P_1, \ldots, P_n$ only occur in this rule (Batory, 2005), we know that no other $P_j, j \neq i$ has been applied. Thus the semantics of *xor* is respected.

3. an *or*-node: mapped to two rules $f : t+$ and $t : P_1|\ldots|P_n$ where some P_i is either n or $[n]$. We follow the expansions of t in the derivation. They must all lead to a P_i, since t only occurs in these two rules. Thus at least one P_i, but perhaps more, occur in the derivation. Therefore, M satisfies the semantics of *or*.

$\square$

However $\mathcal{M}_{BFT1}$ and $\mathcal{M}_{BFT2}$ semantics do not coincide. We are now able to tell what the problem is with Batory's second semantics ($\mathcal{M}_{BFT2}$), and why does it fail to coincide with his first semantics ($\mathcal{M}_{BFT1}$). Batory actually generalised hastily the equivalence that we can notice in the case of *or*-nodes (we abbreviate $f \in m$ by f, m being a model):

- the *or* rule gives that $f \Rightarrow (s_1 \vee \ldots \vee s_k)$,

- the justification rule gives that $\bigwedge_i(s_i \Rightarrow f)$.

Taken together, we thus have an equivalence: $f \Leftrightarrow (s_1 \in m \vee \ldots \vee s_k \in m)$. Unfortunately, this equivalence does not generalise to *xor*. Indeed, the equivalence provided in the *xor* rule ($f \Leftrightarrow$ $\mathtt{choose}_1(s_1, \ldots, s_k)$) results in undesirable behaviours. A simple example where this equivalence fails is given in Figure 9.28 (1) where *a* and *b* are non-primitive *xor*-nodes while B, C, D, E, F are primitive *and$_0$*-nodes.
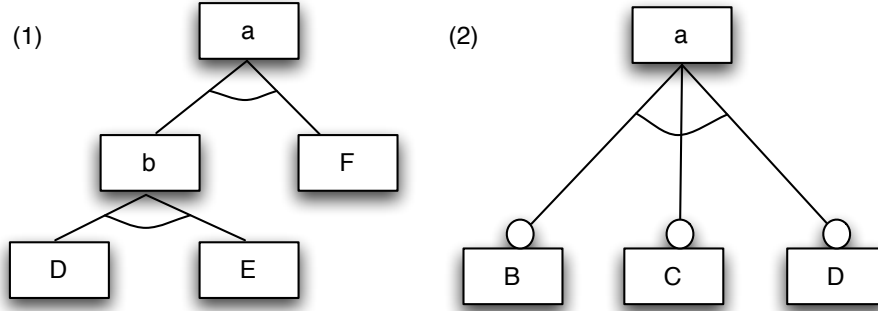
Figure 9.28: BFT with a wrong semantics

FFD semantics, the grammar semantics (Batory, 2005), and common sense all indicate that the meaning of this diagram should be $\{\{D\}, \{E\}, \{F\}\}$. The formula [†] $\mathcal{T}_1(d_1)$ is $a : b|F, b : D|E$ and $\mathcal{T}_2(\mathcal{T}_1(d_1))$ is $(a \Leftrightarrow (b \oplus F)) \wedge (b \Leftrightarrow (D \oplus E))$, that admits the spurious product $\{D, E, F\}$ as abstracted from the model $\{D, E, F, a\}$. Indeed if a is TRUE and F is TRUE then $(a \Leftrightarrow (b \oplus F))$ is TRUE only if b is FALSE. On the other hand if *b* is FALSE and iff *E* and *D* are both TRUE or FALSE then $(D \oplus E)$ is FALSE and $(b \Leftrightarrow (D \oplus E))$ is TRUE. Hence, $\{D, E, F, a\}$ is valid for d_1 propositional formula: $(a \Leftrightarrow (b \oplus F)) \wedge (b \Leftrightarrow (D \oplus E))$.

Another example is illustrated in Figure 9.28 (2) where *a* is a *xor*-node. $\mathcal{T}_1(d_2)$ is $a : [B] \| [C] \| [D]$, and $\mathcal{T}_2(\mathcal{T}_1(d_2))$ is $a \Rightarrow (B \wedge \neg C \wedge \neg D) \vee (\neg B \wedge C \wedge \neg D) \vee (\neg B \wedge \neg C \wedge D)$. This propositional formula means that no two sons of *a* can be TRUE at the same time. However if we study the justification rule for *a* it gives: $B \vee C \vee D \Rightarrow a$ that is not the converse implication, and does not entail it.

Thus, the equivalence mentioned above has been generalised beyond its limits in (Batory, 2005), and its two semantics are different. It is clear that the first one is correct and the second false, that comforts our definition. The tool mentioned in (Batory, 2005) remains applicable to FFD semantics with a slightly different formula in $\mathcal{T}_2$. Indeed we propose to add justification rules to this formula and to replace the wrong equivalences by implications. For rules of the form $r : \pi_1 | \ldots | \pi_k$, we replace the formula $r \Leftrightarrow \mathtt{choose}_1(\pi_1, \ldots, \pi_k)$ by $(r \Rightarrow (\pi_1 \oplus \ldots \oplus \pi_k)) \wedge (\pi_1 \Rightarrow r) \wedge \ldots \wedge (\pi_k \Rightarrow r)$. Hence, $\{D, E, F, a\}$ is not valid anymore for d_1 corrected propositional formula: $(a \Rightarrow (b \oplus F)) \wedge (b \Rightarrow (D \oplus E)) \wedge (D \Rightarrow b) \wedge (E \Rightarrow b) \wedge (b \Rightarrow a) \wedge (F \Rightarrow a)$. The main differences are:

- When *b* is FALSE, the second implication $b \Rightarrow (D \oplus E)$ says that its right part $(D \oplus E)$ can be TRUE or FALSE to set this implication to TRUE;

- The justification rules $(D \Rightarrow b) \wedge (E \Rightarrow b)$ render $\{D, E, F, a\}$ invalid.

[†]$\pi_1 \oplus \pi_2$ is the logical equivalent of $(\pi_1 \wedge \neg \pi_2) \vee (\neg \pi_1 \wedge \pi_2)$, where $\oplus$ is called the *XOR* Boolean operator and corresponds to a $\mathtt{atmost}_1(\pi_1, \pi_2)$

Accordingly Batory has recently provided an erratum to the formula in $\mathcal{T}_2$. For rules of the form $r : \pi_1 | \ldots | \pi_k$, he replaces the formula $r \Leftrightarrow \mathtt{choose}_1(\pi_1, \ldots, \pi_k)$ by $(r \Leftrightarrow (\pi_1 \vee \ldots \vee \pi_k)) \wedge \mathtt{atmost}_1(\pi_1, \ldots, \pi_k)$. Hence, $\{D, E, F, a\}$ is not valid anymore for d_1 corrected propositional formula: $(a \Leftrightarrow (b \vee F)) \wedge \mathtt{atmost}_1(b, F) \wedge (b \Leftrightarrow (D \vee E)) \wedge \mathtt{atmost}_1(D, E)$. This erratum is correct although the justification rules do not appear clearly. Indeed, the two equivalences can be transformed:

- $(a \Leftrightarrow (b \vee F)) \Leftrightarrow ((a \Rightarrow (b \vee F)) \wedge (b \Rightarrow a) \wedge (F \Rightarrow a))$

- $(b \Leftrightarrow (D \vee E)) \Leftrightarrow ((b \Rightarrow (D \vee E)) \wedge (D \Rightarrow b) \wedge (E \Rightarrow b))$

At this point a difference remains between FFD and Batory formulae and concerns the Boolean operators used in the right part of the two implications: in FFD it is a *xor* while in Batory it is an *or*. However this last difference vanishes when Batory uses the $\mathtt{atmost}_1$ operator between b and F and between D and E.

9.4.4 BFT Expressiveness Analysis

Once we have abstracted and corrected BFT's semantics, BFT expressiveness can be studied. Propositional Logic itself is expressively complete and thus BFT. However, BFT without Propositional Logic or constraintless BFT (CBFT) is not expressively complete, see Theorem 9.4.4.

Theorem 9.4.4 *CBFT (Batory, 2005) are not expressively complete.*

Proof The proof is similar to the one provided for the expressiveness of CRFT in Theorem 9.1.11. BFTs cannot express *card*[2..2] among 3 features. We note that when using trees, *and*, *xor*, *or* are associative. So, without loss of generality, we can assume that the first node from the root has two sons s_1, s_2: if he has more, we use associativity to break the list of sons in two. Now each operator imposes its "shape" on the product line. For trees, $[\![s_1]\!], [\![s_2]\!]$ have disjoint primitive features, say without loss of generality $\{A, B, C\}$. $[\![card_3[2..2](s_1, s_2)]\!]$ has none of these shapes.

$\square$

This example of incompleteness would clearly be eliminated by adding "or with cardinality" as suggested in (Batory, 2005) (we call it *card*). Even so, expressive completeness is not reached for CBFT (see Table 9.13).

In (Batory, 2005) this lack of expressiveness is compensated by adding full Propositional Logic as textual constraints. This logic is known to be expressively complete, so that the common part of BFT and CBFT would actually be no longer needed.

9.4.5 BFT Embeddability Analysis

In Section 5.3, we proposed a definition of graphical embeddability (Definition 5.3.9) that generalises the definition of embeddability for context-free languages. Given this definition, we need to look at the abstract syntaxes of FFD and BFT to study their embeddability. The abstract syntax of BFT equals that of GPFT except for textual constraints.

FT languages without *card*-nodes are clearly embeddable into BFT while FT languages with *card*-nodes are not. Conversely, BFT are not embeddable into FT languages where constraint nesting is not allowed. However, if we consider constraintless BFT (CBFT) they are embeddable into all constraintless FTs.

FD languages are not embeddable into BFT since sharing is not allowed in BFT. Conversely, BFT is embeddable into EFD. Indeed, a propositional formula and thus a BFT can be linearly translated into EFD. The idea is that every propositional formula can be translated into an equivalent CNF formula. This translation is linear when auxiliary symbols are allowed in the CNF and is based on logical equivalences: the Double Negative Law, the De Morgan's laws, and the distributive Law. Then this CNF is linearly translated into CRFD according to the translation already provided to prove the complexity of FFD satisfiability (Theorem 9.1.1). In the end, the composition of these translations is linear and therefore BFT is embeddable in EFD but not conversely.

Moreover, if we consider constraintless BFT (CBFT), CBFT is embeddable into RFD, see Table 9.31.

Instead of . . .	**write . . .**	**factor . . .**
opt_1-node	opt_1-node	1
xor_s-node	xor_s-node	1
or_s-node	or_s-node	1
and_s-node	and_s-node	1

Table 9.31: Embedding: CBFT into RFD

9.4.6   BFT Succinctness Analysis

Succinctness (Definition 5.4.1) actually allows comparing the size of the diagram before and after translation. By definition, whenever there is an embedding, there also exists a node-controlled translation. In our case, any FT without *card*-nodes is identically-as succinct as CBFT while any FT with *card*-nodes cannot be translated into CBFT. Conversely, CBFT is as succinct as all FT languages without constraints. If we consider FDs, CBFT can be linearly translated to any FD, see Table 9.31.

Once Propositional Logic is added to CBFT, every FD with only primitive feature can be translated into Propositional Logic and thus BFT. This translation from FD to Propositional Logic corresponds to FFD semantic implementation that is provided in Section 9.1.1. According to the CNF Boolean cardinality encoding proposed by (Sinz, 2005), the expansion factor of the translation of a *card*-node to BF is quadratic. However, once non-primitive features are allowed, the resulting BF should be existentially quantified with non-primitive features. Therefore, the translation from FD to Propositional Logic and thus BFT is exponential.

Conversely, an embedding between BFT (Propositional Logic) and EFD exists. Therefore, this translation is linear and EFD is thus exponentially-as succinct as BFT ((BFT $\leq$ O(2^{EFD})) while BFT is linearly-as succinct as EFD (EFD $\leq$ O(BFT)). As illustrated in Figure 9.29, BFT succinctness results compose with the previous ones.

$O(2^n)$

Propositional
Logic & BFT

$O(n)$

$O(2^n)$ $O(n)$

$O(n)$

(C)OFD   EFD,VFD

$O(n^3)$

$O(n)$ $O(n^3)$ $O(n)$

$O(n)$

BC   RFD

$O(2^n)$ $O(n^3)$

$O(n)$ $O(2^n)$ $O(n)$

vDFD   CBFT
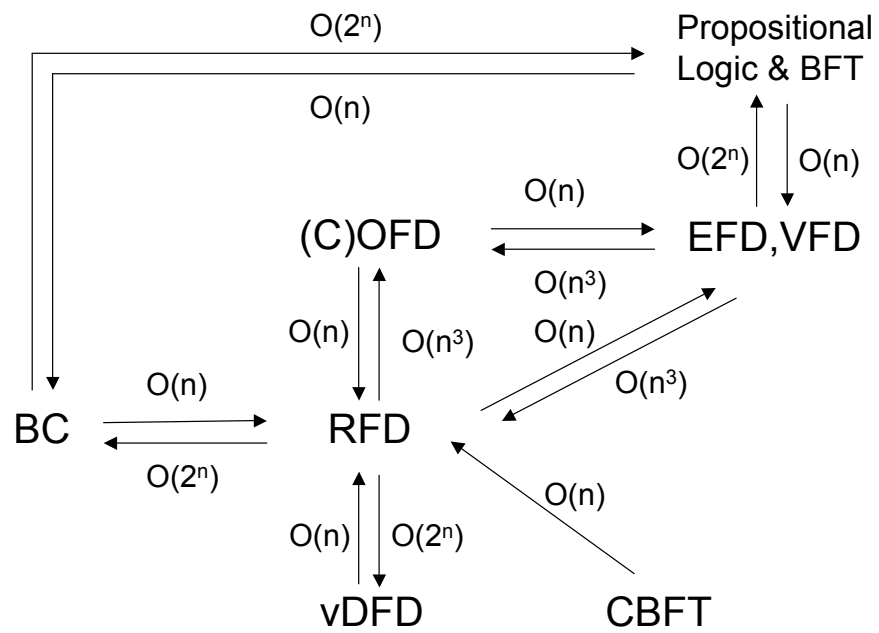
Figure 9.29: BFT Succinctness results

Criteria	OFT	OFD	RFD	EFD	GPFT	PFT	vDFD	BFT	VFD	BC
Definition –Author(s)	(Kang et al., 1990)	(Kang et al., 1998)	(Griss et al., 1998)	(Riebisch et al., 2002)	(Eisenecker and Czarnecki, 2000)	(Eriksson et al., 2005)	(van Deursen and Klint, 2002)	(Batory, 2005)	(Bontemps et al., 2004)	(Shannon, 1938)
–Formal –Semantic Domain	× $\mathcal{PP}(P)$	× $\mathcal{PP}(P)$	× $\mathcal{PP}(P)$	× $\mathcal{PP}(P)$	× $\mathcal{PP}(P)$	× $\mathcal{PP}(P)$	√ $\mathcal{OO}(A)$	√ $\mathcal{PP}(V)$	√ $\mathcal{PP}(P)$	√ $\mathcal{PP}(V)$
Complexity –Satisfiability	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete
–Product– Checking	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	NP-Complete	Linear
–Equivalence	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP$-Complete
–Inclusion	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP^{NP}$-Complete	$coNP$-Complete
–Intersection	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear
–Union	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear	Linear
Expressiveness	Expressively Complete	Expressively Complete	Expressively Complete	Expressively Complete	Expressively Complete	Expressively Complete	Expressively Complete	Expressively Complete	Expressively Complete	Expressively Complete
Embeddability	OFD	RFD	EFD	VFD	RFD	RFD	RFD	EFD	EFD	RFD
Succinctness	FD	OFD	RFD	EFD	FD	FD	FD	Prop. Logic	EFD	BC

Table 9.32: FD Language Analysis

9.5   Chapter Summary

Throughout this chapter, we have compared informal and formal FD languages according to the comparison method proposed in Chapter 7. In the end, we suggest a new FD language called VFD for which (1) the decomposition edges form a DAG and (2) the only allowed node type is *card*-node. One could say that, like our predecessors, we define yet another FD language. However, we argue that our approach is relatively different. Firstly, VFD is not an extension but a simple form of EFD. Secondly, the introduction of VFD is justified since VFD obtains the best scoring according to the comparison method and formal criteria defined previously: VFD is syntactically minimal, yet expressively complete, included in the lowest class of succinctness and it can embed all studied FD language variants (see Table 9.32).

We now summarise our results according to complexity, expressiveness, embeddability, succinctness and semantic equivalence analysis.

Complexity results

The *complexity results* show that most of the decision problems are non-trivial. Since no efficient algorithm exists to solve them, implementations are very likely to face tractability issues in some specific cases. For instance, worst-case execution time for *Product-Checking* grows exponentially with the size of the diagrams in some FD languages. We have shown that the expressively complete FD languages share the same complexity results:

- the *Satisfiability* problem for FDs is *NP-Complete*;

- the *Product-Checking* problem for FDs is *NP-Complete*;

- the *Equivalence* problem for FDs is *coNP^{NP}-Complete*:

- the *Inclusion* problem for FDs is *coNP^{NP}-Complete*;

- the *Intersection* and *Union* problems are *Linear*.

Once a problem is known to be complex, its associated tractability issues could be minimised or at least circumscribed. For FDs, we follow Batory's suggestion (Batory, 2005) to use SAT-solvers (see Chapter 10). Those have the following advantages:

- Many practical problems from various domains reduce to SAT: logic synthesis, model checking, circuit routing, software verification and, of course, FD reasoning;

- SAT-solvers made progress over the years to tackle tractability issues;

- Current SAT-solvers can handle formulae with millions of variables and clauses.

We also mitigate each complexity result according to specific cases for which we determine a lower complexity. For instance, *Product-Checking* for FDs with only *and*-nodes becomes *Linear*. Moreover, we showed that the addition or suppression of some constructs can drastically reduce or increase complexity. As illustrated in Table 9.33, two constructs significantly affect complexity in

FDs. The first one concerns feature sharing. When FDs are restricted to trees with primitive leaves without crosscutting constraints, FD decision problems become trivial. The second one concerns primitive and non-primitive features. When FDs contain only primitive features ($P = N$), *Product-Checking* complexity is reduced from *NP-Complete* to *Linear*, and *Equivalence* and *Inclusion*'s complexity is reduced from $coNP^{NP}$-*Complete* to *coNP-Complete*.

FD	Satisfiability	Product-Checking	Equivalence	Inclusion	Intersection	Union
no restriction	*NP-Complete*	*NP-Complete*	$coNP^{NP}$-*Complete*	$coNP^{NP}$-*Complete*	*Linear*	*Linear*
DAG with $P = N$	*NP-Complete*	*Linear*	*coNP-Complete*	*coNP-Complete*	*Linear*	*Linear*
TREE with leaves $\subseteq P$ and without constraints	*Linear*	*Linear*	*Linear*	*Linear*	*Linear*	*Linear*

Table 9.33: Complexity Analysis

On the one hand, discarding some constructs or abstracting the semantic domain may reduce complexity and tractability issues. On the other hand, it may also hinder the expressiveness of the language or the adequacy of the semantic domain. The relation between expressiveness and complexity is not trivial. Increasing the expressiveness of a language does not necessarily imply that it will increase its complexity. Conversely, reducing the expressiveness of a language does not necessarily imply that it will reduce its complexity. Even if it is often the case.

Expressiveness results

The *expressiveness results* indicate that all studied FD languages are expressively complete. They also identify which constructs are necessary to guarantee expressive completeness and which are not. For instance, we have seen that without constraints, tree-shaped languages are not expressively complete. Therefore, expressiveness justifies some extensions brought to OFT (Kang et al., 1990) but not all of them. Indeed, simple tree-shaped languages are usually expressively incomplete when constraints are not allowed. In particular, they cannot express the choice of two features among three. In more details, expressiveness:

- justifies the proposal by (Kang et al., 1998) (OFD) to transform tree-shaped language to DAG-shaped language. Indeed, OFD is expressively complete with or without constraints. A fortiori, all further FD languages that are DAGs and allow feature sharing (RFD, EFD, VFD) are expressively complete;

- does not justify the proposal by (Griss et al., 1998) (RFD) to add the *or* operator to OFT. Even with the *or* operator a simple constraintless tree-shaped language does not attain expressive completeness. RFD is still unable to express the choice of two features among three;

- does not justify the proposal by (Riebisch et al., 2002) (EFD) to use the *card* operators. However, both (Griss et al., 1998) and (Riebisch et al., 2002) allow DAGs and are therefore expressively complete;

- justifies the proposal by (Bontemps et al., 2004) (VFD) to use only the *card* operators within feature DAGs. Indeed, feature sharing and *card*-operators are sufficient to assure expressive completeness;

- justifies the proposal by (Batory, 2005) (BFT) where Boolean constraints are added to bring expressive completeness. Obviously, adding Propositional Logic to tree-shaped languages brings expressive completeness.

Embeddability results

The *embeddability results* indicate which FD languages can be translated into which one while preserving the original diagram semantics and structure. Preserving the meaning means that both diagrams, the original and its translation will map the same element in the common semantic domain. Preserving the structure means that the translation generates only a linear increase in size between the original and translated diagram. According to the definition of embedding (Definition 5.3.1), we have proved that:

- FTs are embeddable into RFD but not conversely;

- RFDs are embeddable into EFD but not conversely;

- OFDs are embeddable into RFD but not conversely;

- BCs are embeddable into EFD but not conversely;

- vDFDs are embeddable into EFD but not conversely;

- BFTs are embeddable into EFD but not conversely;

- VFDs are embeddable into EFD and conversely.

In addition, these embeddability results compose. Indeed, if FTs are embeddable into RFD and RFDs are embeddable into EFD then FTs are embeddable into EFD. Accordingly, OFDs are embeddable into VFD and EFDs are not embeddable into BFT, etc. Once these compositions are applied, it appears that all FD languages are embeddable into EFD and VFD. However, the embeddability analysis also indicates that EFD is self-embeddable. This means that EFD is unnecessarily complex because all constructs in EFD are easily definable using only one of them: *card*-nodes. These constructs are said to be redundant. This justifies why we suggest to prefer VFD that contains only *card*-nodes. VFD is therefore minimal and still in the same class of succinctness than EFD. Redundancy and succinctness are also related: languages often contain redundant constructs to improve their succinctness. One could question whether it is preferable to use non-redundant languages or more succinct languages. The answer depends on the context of use of the language.

Succinctness results

The *succinctness results* are summarised in Figure 9.29 and indicate the existence of essentially five classes of succinctness for FDs:

1. Propositional Logic;

2. BCs are exponentially-as succinct as Propositional Logic due to the sharing allowed in BC while not in Propositional Logic;

3. OFDs are exponentially-as succinct as BC due to the introduction of primitive and non-primitive features which implies that the corresponding BF should be existentially quantified;

4. RFDs are cubically-as succinct as OFD, due to the use of *or*-nodes;

5. EFDs are cubically-as succinct as RFD, due to the use of *card*-nodes.

The succinctness results presented in Figure 9.29 compose. For instance, we know that in the worst case the translation from vDFD to OFD increases the size of the original diagram cubically. On the contrary, the translation from EFD to VFD causes no increase in size. The translation from VFD to EFD multiplies in the worst case the size of the original diagram linearly. EFD and VFD are thus included in the same class of succinctness (called EFD). We also observe that the translation from any FD language to Propositional Logic or BC may increase the size of the diagram exponentially since they should be existentially quantified.

Semantic Equivalence results

The comparative semantic analysis of FFD compares FFD with three formal languages: BC, vDFD and BFT. First, we redefined these languages according Harel and Rumpe's principles (Chapter 4). Then we studied whether there exists a relation (abstraction function) between their respective semantic domains. Finally, we determined whether they are semantically equivalent according to this abstraction function. We conclude that:

- BC is semantically equivalent to FFD modulo an abstraction function. This abstraction function consists in discarding the non-primitive features from BC's semantic domain.

- vDFD is not semantically equivalent to FFD although there exists an abstraction function between their semantic domains. The difference originates from the definition of their respective semantic functions. Indeed, vDFD follows an edge-based semantics while FFD follows a node-based semantics. However, we have been able to obtain a semantic equivalence between FFD and vDFD by applying a preliminary transformation to the abstract syntax of FFD in order to behave as an edge-based semantics.

- The two semantics of BFT provided in (Batory, 2005), namely the grammar and propositional semantics, are not semantically equivalent. The difference originates from the definition of their semantic functions. Indeed, the propositional semantics defined in (Batory, 2005) generalised equivalence too much. However, we determined that FFD and the grammar semantics defined in (Batory, 2005) are semantically equivalent.

- For both BFT and vDFD, the comparative semantic analysis led to identify some minor unintended errors or omissions in the original definitions.

When these results are composed, a semantic domain category is created for FDs. As illustrated in Figure 9.30, these semantic domains are related by abstraction functions ($\mathcal{A}_1, \mathcal{R}, \mathcal{A}$). This category can be extended with new comparative analyses and other compositions of abstraction functions.
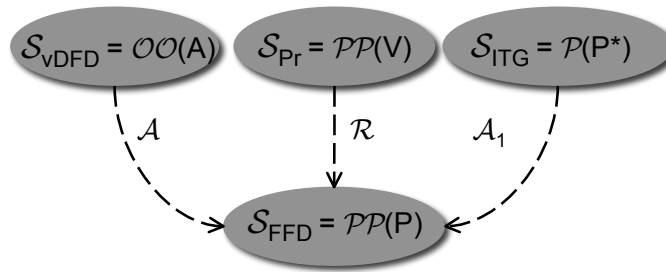

Figure 9.30: Semantic Domains Network

In the next chapter, we will describe a reasoning tool supporting VFD and its implementation based on a SAT-solver.

Part IV

Automation of Feature Diagram Languages

Chapter 10

VFD: A Reasoning Tool

I n the previous chapter, the FD language quality analysis suggests VFD as the language currently obtaining the best ranking according to the studied criteria. VFD's abstract syntax has been defined in Table 8.1 and Section 8.2.2.1 while VFD's semantics is defined in Section 8.2.2.3. Formalisation of VFD is crucial to have a clear and concise definition of VFD, to avoid ambiguities and to define decision problems on VFD. Since these decision problems are formally defined and their complexity studied (Section 9.1.1), they can be automated and efficient tool support can be provided. Accordingly, the purpose of this chapter is to describe a tool that supports feature modelling and reasoning on FDs written in VFD.

The structure of this chapter is as follows. First, in Section 10.1 we summarise the expected functionalities for a feature modelling and reasoning tool. Then, we present the design of our tool (Section 10.2) with a focus on the implementation of VFD's semantics (Section 10.3). Afterwards, the mobile phone example is used in Section 10.4 to illustrate usage of the tool. Finally, in Section 10.5 we present various components reused during the implementation of the tool.

10.1 Functionalities

A useful SPL modelling environment requires some tool support to facilitate feature modelling, reasoning and interoperability. This tool support should provide five categories of functionalities:

- *Visualisation of Feature Diagrams*. The tool should provide a convenient visualisation for FDs. Currently, the only one visualisation is available in the tool. VFD is not supposed to be an end-user language. No concrete syntax has thus been defined and the tool offers to visualise in a very basic form the VFD's abstract syntax (Figure 10.6). However, nothing forbids to define a concrete syntax for VFD and to use it as an end-user language. This is not a trivial task and we strongly advice to follow guidelines when defining concrete syntaxes, such as Moody's principles (Moody, 2006b). Those are meant to guarantee the cognitive effectiveness of diagrams. According to the SEQUAL framework (Section 3.1), these principles mainly concern the empirical quality of models that is out of this thesis' scope.

- *Editing Feature Diagrams*. The tool should provide functionalities that enable to edit FDs. The manipulation of the various graphical constructs presented in the abstract syntax should

be facilitated. The user should be able:

- to add, edit or remove nodes. Where nodes could be primitive or non-primitive nodes,

- to edit node names and cardinalities,

- to add, edit or remove edges between nodes,

- to add or remove constraints such as *requires* or *excludes*,

- to save FDs,

- to load FDs.

- *Reasoning on Feature Diagrams.* The tool should provide functionalities to reason on FDs. Indeed, once FD semantics has been formally defined, various FD checks and transformations are applicable:

  - FD Satisfiability (Definition 7.1.1),

  - FD Membership or Product checking (Definition 7.1.2),

  - FD Equivalence (Definition 7.1.3),

  - FD Intersection (Definition 7.1.4)

  - FD Inclusion (Definition 7.1.5),

  - FD Union (Definition 7.1.6),

  - Product Listing: List all the valid products contained in the SPL. The valid products are given by FFD semantic function (Definition 8.2.3),

  - Dead Feature Detection: A feature is dead when no product in the PL includes it (Definition 10.1.1).

    Definition 10.1.1 (Dead feature) *A feature f is* dead *for an (O)FD d when:*
    $\nexists p \in \mathcal{M}_{FD}[\![d]\!] \cdot f \in p$

    Example 10.1.1 *For instance, if we consider the OFD illustrated in Figure 8.4, its semantics is composed of two products:* $\{\{f_1, f_2, f_3, f_4, f_7, f_8, f_9\}, \{f_1, f_2, f_3, f_5\}\}$. *According to this semantics, we notice that the feature* (f_6) *is not included in any product. Hence,* f_6 *is a* dead *feature.*

  - Common Feature Detection: A feature is common when all products in the SPL include it (Definition 10.1.2).

    Definition 10.1.2 (Common feature) *A feature f is* common *for an (O)FD d when:*
    $\nexists p \in \mathcal{M}_{FD}[\![d]\!] \cdot f \notin p$

    Example 10.1.2 *For instance, if we consider the OFD illustrated in Figure 8.4, its semantics is composed of two products:* $\{\{f_1, f_2, f_3, f_4, f_7, f_9\}, \{f_1, f_2, f_3, f_5\}\}$. *According to this semantics, we notice that three features* (f_1, f_2, f_3) *are included in all valid products. They are* common *features.*

- *Persistence of Feature Diagrams.* The tool should allow making FDs persistent. Both the layout and the content of the FD should be made persistent. In addition, it should be saved under a formal description suitable for further processing. In our case, we have selected the XML format.

- *Interoperability between Feature Diagrams.* In Chapter 9 we have seen that VFD is expressively complete and that embeddings (translations) between VFD and every language covered by FFD exist. Hence, VFD may serve as a pivot language to facilitate interoperability between FD languages. Accordingly, the next evolution of the tool will be to offer functionalities that allow translating any FD into VFD and then translating back (if possible) this VFD into other FD languages preserving the original semantics of the FD. These translations will be based on the embeddings, translations and abstraction functions provided throughout this work. Solutions will also be needed to correctly manage the different layouts of the models. In the end, the tool will help reasoning on FDs written with different languages.

10.2   Three-Tiers Architecture

The architecture of the tool (see Figure 10.1) is a three-tiers architecture. A three-tiers architecture is a software architecture framework in which the user interface, the business logic and the data management are developed and maintained as independent modules. These three modules or tiers are called respectively: the *presentation tier*, the *application tier* and the *data tier*.

For the *presentation tier*, our tool offers two possibilities. The first alternative is to use *graphviz* library (AT&T, 2007) and its *DOT* language (Section 10.5.2) to visualise FDs. The second alternative is to use the *Graphical Modelling Framework (GMF)* (IBM and Borland, 2007) to automatically generate a FD graphical editor (Section 10.5.4).

For the *application tier*, FD concepts are represented in a *data model* and are used as inputs for various components such as editor, translator and reasoner:

- The *editor component* manipulates these concepts and uses a *graph component* that gathers classical graph algorithms based on matrix transformations. Hence, successors and predecessors could be easily identified and research in graphs is facilitated. In addition, configuring a FD necessitates to eliminate variation points (nodes) and thus to use graph transformations.

- The *translator component* translates the FD into various formats. For instance, a FD can be translated into different FD languages or into a BF in CNF that serves as input for the reasoner component.

- The *reasoner component* is based on our FD semantics and thus needs to have full access to a SAT-solver. In our case, *Sat4j* (Berre and Parrain, 2007) is used (Section 10.5.1). We have defined the architecture in order to minimise the changes if the SAT-solver's implementation needs to be changed.

For the *data tier*, the first alternative is to use the *Xstream component* (Section 10.5.3) that allows serialising every object into XML. Concretely, every edge is represented by a couple of nodes and

*Presentation
Tier*

DOT

Editor

input

FD

input

Translator

<<uses>>

CNF

input

Graph

<<uses>>

<<uses>>

Reasoner

SATSolver:
SAT4j

*Application
Tier*

<<uses>>

Object
serialization:
XStream

*Data
Tier*

XML

DataBase
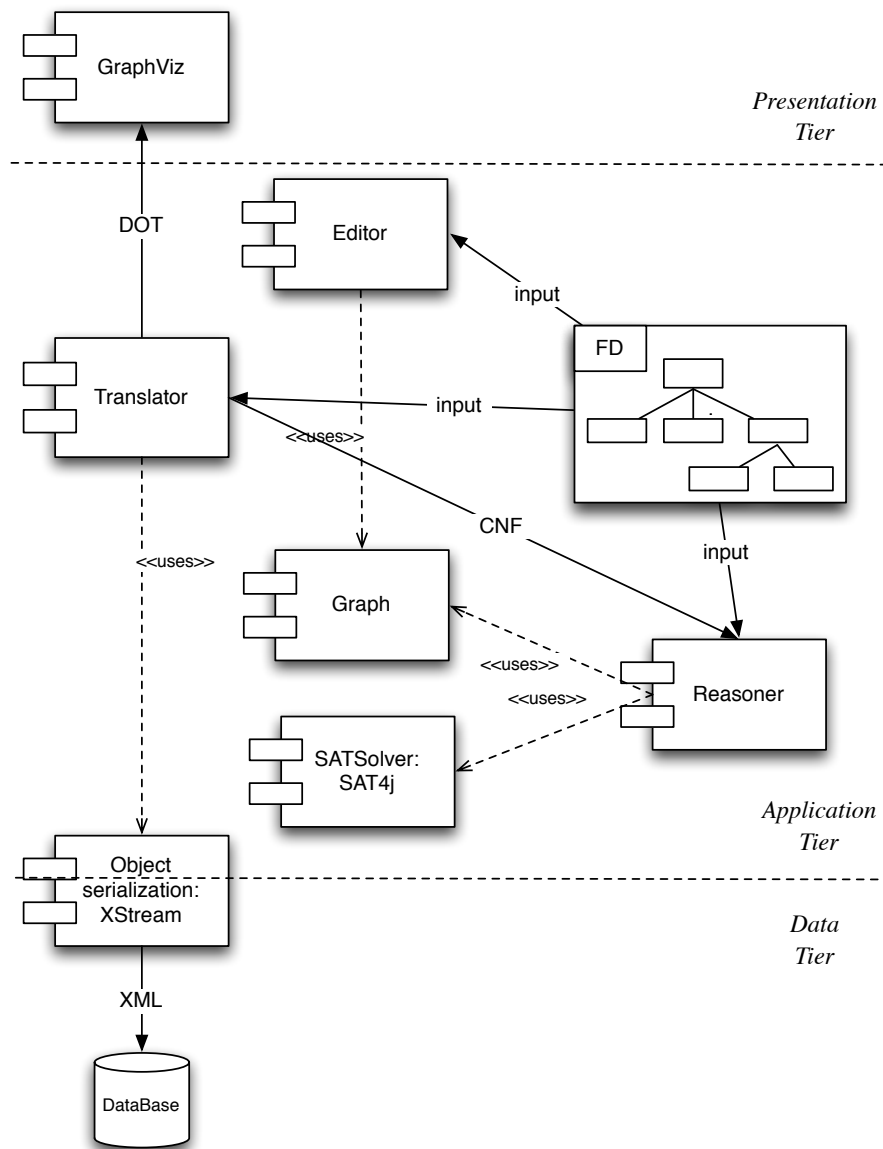
Figure 10.1: Tool Architecture

saved into an XML format that can be easily parsed to restore a copy of the original FD. The second
alternative is to use the persistence functionalities offered by the edito generated with GMF.

10.3   Application Tier

The application tier is the core of the software since it contains its logic. All the aspects concerning the graphical interface and the data storage are left to the other tiers. In this section, first the FD Data-Model is presented (Section 10.3.1). Once the main concepts of the FD Data-Model are defined, other classes are described to edit a FD (Section 10.3.2), to translate it (Section 10.3.3) and to reason on it (Section 10.3.4). Moreover, in Section 10.3.3.1 we present and illustrate the details of our implementation of FD's semantics based on a translation from VFD to BF.

10.3.1   FD Data-Model

As mentioned before, a FD is represented by a DAG composed of *Nodes* and *Edges* (see Figure 10.2). A FD represents a set of *Products* that are composed of *Nodes*. A non-terminal node is considered as a variation point to which we may associate a question. For each decomposition edge of this variation point, we may associate an answer to the corresponding question. These questions and answers help determining which decisions should be taken. A *Decision* is associated to a variation point and determines which of its sub-features are selected. This selection is constrained by the variation point cardinalities. A *Configuration* is a set of *Decisions* that should be complete to derive one final product from the SPL. A configuration is complete when there exists a valid decision for each variation point.

Figure 10.2 contains a class diagram representing the main concepts of VFD. This class diagram is complemented in Figure 10.3 by other classes dedicated to the handling of these concepts (FD_Editor), their translation (FD_Translator) and the reasoning support associated to them (FD_Reasoner).

10.3.2   FD_Editor

The class FD_Editor gathers the various functionalities that enable to create, modify or initialise a FD. The main functionalities are to add the root, to add and remove nodes, to add and remove edges, to add and remove *excludes* and *requires* constraints, to load a FD from a XML file.

10.3.3   FD_Translator

The class FD_Translator is used to translate the FD's Structure into various formats. Mainly three formats are used: DOT, XML and CNF. The DOT language is a plain text graph description language. It is an intuitive and simple way for describing graphs. A Boolean logic formula is in CNF, if it is a conjunction of clauses, where a clause is a disjunction of literals. CNF is used as input format by most SAT-solvers. This class could also integrate translations from on FD language to another following the translations and embeddings provided throughout the thesis.

Our choice in the implementation of the FD semantics is to provide a translation from VFD's abstract syntax to BF. Then, these BFs are converted to CNF which is then passed to a SAT-solver to check its satisfiability. The SAT-solver aims to determine if the variables of a given CNF can be assigned in such a way that they make the CNF evaluate to TRUE. Then, the SAT-solver returns

Figure 10.2: FD Data-Model

all the possible variable assignments that make the CNF evaluate to TRUE. If no such assignment exists then the BF is unsatisfiable. In addition, the SAT-solver can also identify contradictions in the BF that make it unsatisfiable.

The translation from VFD's abstract syntax to BF is not trivial especially with Boolean Cardinalities. Indeed, an optimal CNF Encoding of Boolean Cardinality is necessary. The translation presented in Section 10.3.3.1 reuses optimal encodings from (Sinz, 2005) (Section 10.3.3.2).

10.3.3.1   VFD to BF

The translation from a VFD to a BF is node controlled. each *card*-node is translated into a BF according to its cardinalities, its son(s), its parent(s) and the justification rule. Constraints such as *requires* and *excludes* between nodes can directly mapped to a BF. More complex constraints could also be added by providing the adequate BF, therefore requiring no mapping.

As illustrated in Figure 10.4, each node in our abstract syntax corresponds to a node of type *card* with a name (say g) and a minimum and maximum cardinalities (i, j). Each node can have several sons, $f_1 \ldots f_n$, and be shared by several parents, $p_1 \ldots p_m$. Therefore, the node g is a *card* node

Figure 10.3: FD Handling

of arity n and cardinalities (i, j), noted $card_n[i..j]$. In addition, the cardinalities should respect the constraints: $0 \leq i \leq j \leq n$. Based on these concepts, we can now describe the translation from VFD to BF in Table 10.1[*]. This translation is based on an optimal CNF encoding of Boolean cardinality constraints among several encodings proposed in (Sinz, 2005). We present and justify the one we have selected ($LT_{SEQ}^{n,j}$) in Section 10.3.3.2.

According to the justification rule, at least one of the parents ($p_1 \ldots p_m$) of each node evaluated to TRUE should be also evaluated to TRUE, except for the root that by definition has no parent and is always evaluated to TRUE. Hence, the final result of the translation of a *card*-node is the conjunction of the corresponding BFs from Tables 10.1 and 10.2.

Finally the BF are translated into the standard input for every SAT-solver: DIMACS CNF. As illustrated in the following CNF file, the input starts with comments (each line starts with c). The

---

[*]Formulae in Table 10.1 are not yet in CNF. This is for readability reasons. They are converted to CNF with a standard CNF conversion algorithming.
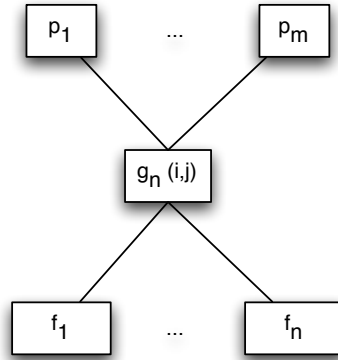
Figure 10.4: A VP Node

$\lambda(g)$	Conditions	Conjunctive Normal Form
$card_n[0..0]$	$n \geq 1$	$\bigwedge_{i=1..n} (\neg g \vee \neg f_i)$
$card_n[0..n]$		no output generated
$card_n[0..j]$	$1 \leq j < n, n \geq 2$	$\neg g \vee LT_{Seq}^{n,j}(f_1, \ldots, f_n)$
$card_n[1..n]$	$n \geq 1$	$\neg g \vee f_1 \vee \ldots \vee f_n$
$card_n[n..n]$	$n \geq 2$	$\bigwedge_{i=1..n} (\neg g \vee f_i)$
$card_n[i..j]$	$1 \leq i \leq j < n, n \geq 2$	$\neg g \vee (GT_{Seq}^{n,i}(f_1, \ldots, f_n) \wedge LT_{Seq}^{n,j}(f_1, \ldots, f_n))$
f_1 requires f_2		$\neg f_1 \vee f_2$
f_1 excludes f_2		$\neg f_1 \vee \neg f_2$

Table 10.1: VFD to CNF

number of variables (*nbvar*) and the number of clauses (*nbclauses*) is defined by the line "p cnf *nbvar nbclauses*". Afterwards, each line specifies a disjunctive clause. In this clause, a positive literal is denoted by indices beginning at 1 and negative literals are denoted by negative indices. A zero indicates the end of a disjunctive clause. For instance, the BF $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$ is translated into this CNF:

```
c A sample .cnf file.
p cnf 3 2
1 -3 0
2 3 -1 0
```

VFD Nodes	Justification rule in CNF
$card_n[i..j] = root$	g
$card_n[i..j] \neq root$	$\neg g \vee p_1 \vee \ldots \vee p_m$

Table 10.2: BF for justification rule

Checking the satisfiability of a CNF formula will provide all the possible sets of values for the variables that evaluate this formula to TRUE. Once the values associated to the auxiliary variables generated by the encoding are removed, each set of values corresponds to a configuration. However, configurations should not be confused with products. Indeed, non-primitive features are not relevant within our semantic domain and several configurations could map to the same product. To remove these non-primitive features, one solution is to quantify existentially the resulting BF with non-primitive features. In our implementation, we have make another choice to avoid the manipulation of EQBF within SAT-solvers. Instead we leave the formula as it is and use it directly as input for a classical SAT-solver. The generated output will be the set of all acceptable configurations. Then, we apply a projection on this set to eliminate (1) all auxiliary variables generated by the CNF encoding of Boolean Cardinality (see Section 10.3.3.2), (2) all non-primitive features and (3) all duplicate products. The resulting set will be the valid set of products.

10.3.3.2   An Optimal CNF Encoding of Boolean Cardinality

Boolean minimal and maximal cardinality constraints (i, j) are formulae expressing that at least i ($\geq i(x_1, ..., x_n)$) and at most j ($\leq j(x_1, ..., x_n)$) out of n propositional variables are TRUE. In (Sinz, 2005), the author proposed two CNF encodings for the ($\leq j(x_1, ..., x_n)$). He also studies the complexity of these encodings according to the number of clauses and auxiliary variables generated.

Since complexity is a major issue in FD reasoning, an efficient encoding is crucial. The first encoding proposed in (Sinz, 2005) uses a parallel counter and requires only $7n$ clauses and $2n$ auxiliary variables. The second encoding he proposes uses a sequential counter and requires $O(n.j)$ clauses and $O(n.j)$ auxiliary variables. The advantage of the first encoding is that it will generate less clauses. The advantages of the second encoding is that (1) it is good for small values of j and that (2) inconsistencies can be detected by the SAT-solver in linear time by unit propagation. Therefore, the SAT-solver can detect efficiently that the cardinality constraints are violated. In VFD, as the number of possible variants is limited, the values of j are often small. Hence, we select the encoding that uses a sequential counter ($LT_{SEQ}^{n,j}$) to transform our VFD nodes into CNF.

For instance, the sequential encoding of the constraint $\leq j(x_1, \ldots, x_n)$ where $j = 2$, $n = 3$ and indices of input variables are { 1, 2, 3 } will result in a CNF formula with 7 variables and 8 clauses. The encoding generates 4 auxiliary variables { 4,5,6,7}:

```
p cnf 7 8
-1 4 0
-5 0
-2 6 0
-4 6 0
-2 -4 7 0
-5 7 0
-2 -5 0
-3 -7 0
```

Finally, to encode the minimal constraint $\geq i(x_1, \ldots, x_n)$ the idea is to reuse the ($LT_{SEQ}^{n,j}$) encoding noting that $\geq i(x_1, \ldots, x_n)$ is equivalent to $\leq (n - i)(\neg x_1, \ldots, \neg x_n)$. We call this encoding ($GT_{SEQ}^{n,i}$).

10.3.4   FD_Reasoner

The class FD_Reasoner gathers the various functionalities that allow reasoning on FDs. The main functionalities are to get the semantics (or list all products), to eliminate variability according to a decision or a complete configuration, to check the membership of a product, to check FD satisfiability, to check equivalence between two FDs, to check inclusion of one FD into another, to check the FD for dead and common features.

10.4   Mobile Phone example

The mobile phone example has been already detailed in Section 2.1.2. Mobile phone systems are embedded systems that offer various features: voice communication, text messaging, phonebook, calendar, camera, internet browsing, games, etc. These features may vary from one product to another and dependencies between them can be modelled in a FD. In Figure 10.5, we reproduce the FD that describes a simplified product line of mobile phone systems using OFD (Kang et al., 1998).
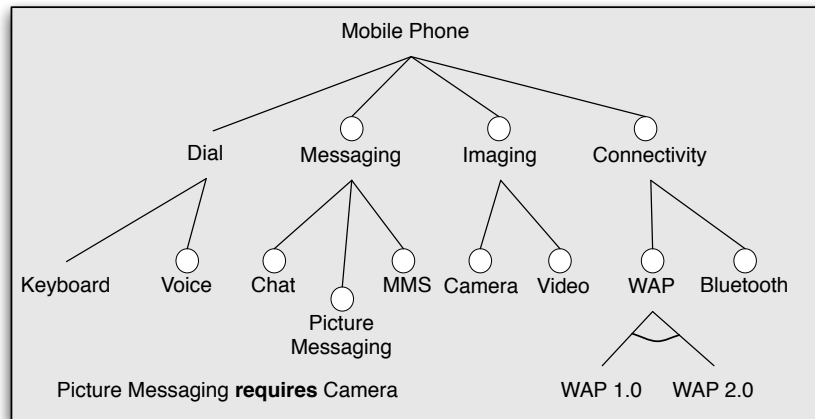
Figure 10.5: FORM FD: Mobile Phone PL

This OFD (Figure 10.5) can be translated into its corresponding VFD (Figure 10.6[†]) according to the embeddings defined in Tables 9.16 and 9.14. Each node is translated into *card*-node(s) with the appropriate arities and cardinalities. For instance the optional and *and*-node *Messaging* is translated into two nodes *OptMessaging* of arity 1 and *Messaging* of arity 3 with respectively cardinalities $(0, 1)$ and $(3, 3)$ and one edge from *OptMessaging* to *Messaging*.

Once the FD is in VFD, the tool translates each *card*-node into its corresponding BF in CNF according to the translations provided in Tables 10.1 and 10.2. For instance, the node *Messaging*$(3, 3)$ is translated into the following BF where the last disjunction corresponds to the justification rule:

[†]The layout for this VFD is the one provided within the tool
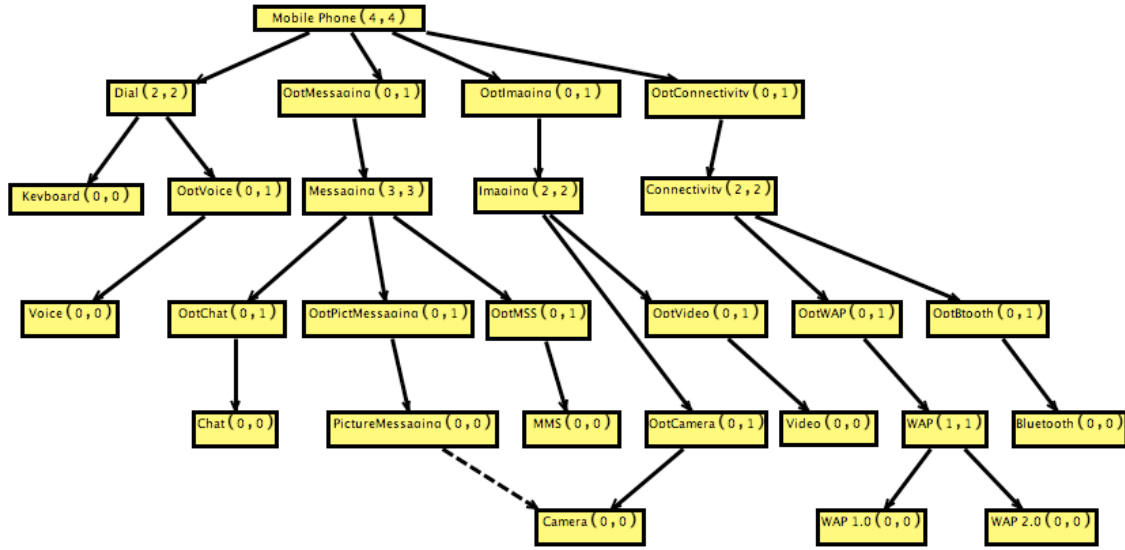
Figure 10.6: VFD Abstract Syntax: Mobile Phone PL

($\neg$ *Messaging* $\vee$ *OptChat*) $\wedge$
($\neg$ *Messaging* $\vee$ *OptPictMessaging*) $\wedge$
($\neg$ *Messaging* $\vee$ *OptMMS*) $\wedge$
($\neg$ *Messaging* $\vee$ *OptMessaging*)

Once all nodes have been translated it remains to translate the crosscutting constraints. Hence, the *requires* constraint between *PictureMessaging* and *Camera* is translated into one disjunction: $\neg$ *PictureMessaging* $\vee$ *Camera*.

The conjunction of these BFs in CNF is then passed to the SAT-solver. Hence, we are now able to reason on VFD.

Once the BF[‡] in CNF, corresponding to the OFD, has been solved by a SAT-solver, the tool applies a projection on its output. This projection essentially eliminates auxiliary symbols that are non primitive features (*P*). The output of this projection is the set of valid products corresponding to the OFD. When *P* includes all nodes, the OFD represents a SPL with 462 valid products. When *P* only includes the leaf features, the FD represents a SPL with still 288 valid products. The complete list of 288 products can be found in Appendix I. Here are some of them, each described as a set of primitive features:

1. {*Keyboard*}

2. {*Keyboard*,*Voice*}

3. {*Keyboard*, *Picture Messaging*, *Camera*}

[‡]The complete BF for this example can be found in Appendix I.

4. {*Keyboard*, *Voice*, *MMS*, *Picture Messaging*, *Chat*, *Camera*, *Video*, *WAP* 2.0, *Bluetooth*}

5. ...

If we had an additional *excludes* constraint between the nodes *Dial* and *Chat*, the resulting number of products would be reduced to 144 (Figure 10.7). When a common feature excludes an optional feature, all the products containing this optional feature become invalid. Therefore, all Mobile Phones with *Chat* are not allowed anymore. Hence, the total number of products is divided by two. This constraint is purely illustrative and irrelevant in the domain.
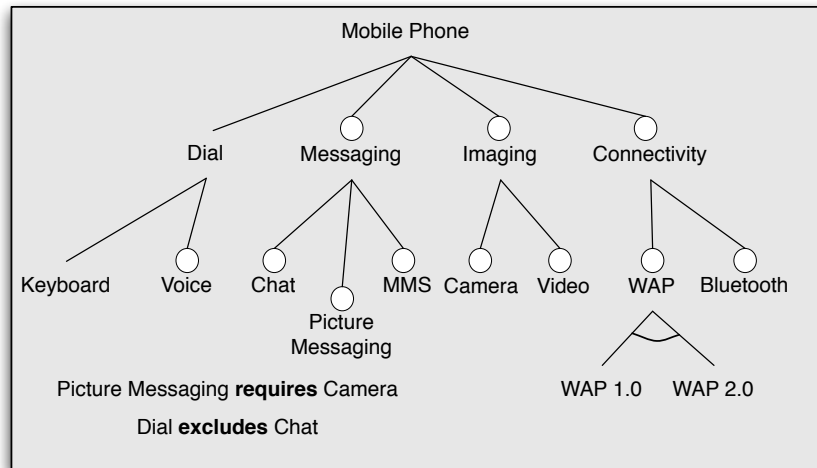
Figure 10.7: FORM FD: Mobile Phone PL with additional constraint

Once we have access to the SAT-solver, the resolution of the decision problems relative to VFD are facilitated. For the OFDs given in Figures 10.5 and 10.7 the tool produces the following results:

1. *FD Satisfiability*: Both OFDs are satisfiable.

2. *Product Listing*: 288 products are listed for the first OFD (Figure 10.5) and 144 for the second (Figure 10.7).

3. *Product Checking*: Checking the membership of the product {*Mobile Phone*, *Dial*, *Keyboard*, *Messaging*, *Chat*, *Picture Messaging*, *Imaging*, *Camera*} reveals that it is a valid product for the first OFD (Figure 10.5) but not for the second (Figure 10.7) since *Dial excludes Chat*.

4. *FD Equivalence*: the first OFD (Figure 10.5) is equivalent to itself but is not equivalent to the second one (Figure 10.7).

5. *FD Inclusion*: the second OFD (Figure 10.7) is included into the first one (Figure 10.5) but not conversely.

6. *Dead Features*: the first OFD (Figure 10.5) contains no dead features while the second OFD (Figure 10.7) contains one dead feature: *Chat*, since *Dial* is mandatory and excludes *Chat*. *Chat* feature will thus never appear in any product. Hence, detecting dead features can help debugging overconstrained FDs.

7. *Common Features*: the first OFD (Figure 10.5) contains three common features: *MobilePhone*, *Dial* and *Keyboard*, while the second (Figure 10.7) contains only one: *Keyboard*.

10.5   Reused Components

In this section, we introduce the different components that have been reused in the development of the VFD reasoning tool. The heart of the tool is based on the SAT4J library (Section 10.5.1) which provides the adequate algorithms to solve satisfiability problems. Another library called Graphviz (Section 10.5.2) is used to generate a graphical representation of FDs based on graph visualisation techniques. The Xstream library (Section 10.5.3) is used to guarantee the persistence of FDs by serialising the object representation of FDs into an XML format that can be easily exchanged. Finally, we have used the Graphical Modelling Framework (GMF) (Section 10.5.4) to generate a tool offering standard support for creating, saving and editing VFD diagrams.

10.5.1   SAT4J

The SAT4J (Berre and Parrain, 2007) project aims to provide an efficient library of SAT-solvers in Java. As our FD semantics is defined in terms of BF, we translate each FD into a CNF respecting the DIMACS CNF format. The algorithms and the global approach on which SAT4J relies are described in (Eén and Sörensson, 2004).

10.5.2   Graphviz

Graphviz (AT&T, 2007) enables to visualise structural information as diagrams of abstract graphs and networks. Since FDs are DAGs we use this open source software to easily visualise them. The Graphviz layout program takes descriptions of graphs in a simple textual language, and generates diagrams in several formats: images and SVG for web pages and postscript for inclusion in PDF or other documents. It also allows displaying graphs in an interactive browser. In Graphviz, the layout of the graph can be customised according to colours, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes.

   To interact with Graphviz, we produce as input a FD translated in DOT and we receive as an output a postscript that we display in a graph browser. The abstract syntax of the DOT language is defined in Table 10.5.2. Terminals are emphasised in bold font. Literal characters are given in single quotes. Parentheses "(" and ")" indicate grouping when needed. Square brackets "[" and "]" enclose optional items. Vertical bars "| "separate alternatives.

   For instance, the abstract syntax of our illustrative example is generated automatically by graphviz within a jpg (see Figure 10.8). The abstract syntax of the FD is translated into a DOT expression that is computed by Graphviz to visualise the FD. The complete DOT expression corresponding to the mobile phone example can be found in Appendix I.

graph	:	[**strict**] (**graph** — **digraph**) [ID] '{' stmt_list '}'
stmt_list	:	[stmt [';'] [stmt_list]]
stmt	:	node_stmt
			edge_stmt
			attr_stmt
			ID '=' ID
			subgraph
attr_stmt	:	(**graph** — **node** — **edge**) attr_list
attr_list	:	'[' [a_list] ']' [attr_list]
a_list	:	ID ['=' ID] [','] [a_list]
edge_stmt	:	(node_id — subgraph) edgeRHS [attr_list]
edgeRHS	:	- - (node_id — subgraph) [edgeRHS]
node_stmt	:	node_id [attr_list]
node_id	:	ID [port]
port	:	':' ID [':' compass_pt]
			':' compass_pt
subgraph	:	[**subgraph** [ID]] '{' stmt_list '}'
			subgraph ID
compass_pt	:	(**n** | **ne**| **e** | **se**| **s**| **sw**| **w** | **nw**)

Table 10.3: The DOT Grammar

10.5.3 XStream

XStream (Walnes and Schaible, 2007) is a component that allows serialising objects to XML and back again. The translation from FD to XML provides persistence and a formal description for FD in a format suitable for further processing. Briefly, the XML format for a FD is the root plus the list of edges where each edge is a couple of nodes. The complete XML version of the mobile phone example can be found in Appendix I.

10.5.4 GMF

The visualisation of FDs with Graphviz is convenient. Nevertheless, it does not allow FD graphical edition. All modelling tools usually include a graphical editor in which each concept can be edited, resized and manipulated. A convenient solution is to use the Graphical Modelling Framework (GMF) (IBM and Borland, 2007).

  GMF is a model-driven framework relying on the Eclipse platform. This framework aims to facilitate the development of graphical editors in a generative manner based on the Eclipse Modelling Framework (EMF) and Graphical Editing Framework (GEF). The editor development with GMF is divided into six activities:

  1. Develop Domain Model. This model is a meta-model the language. It is based on the FD Data-Model illustrated in Figure 10.2 and defines the *abstract syntax* of the language.
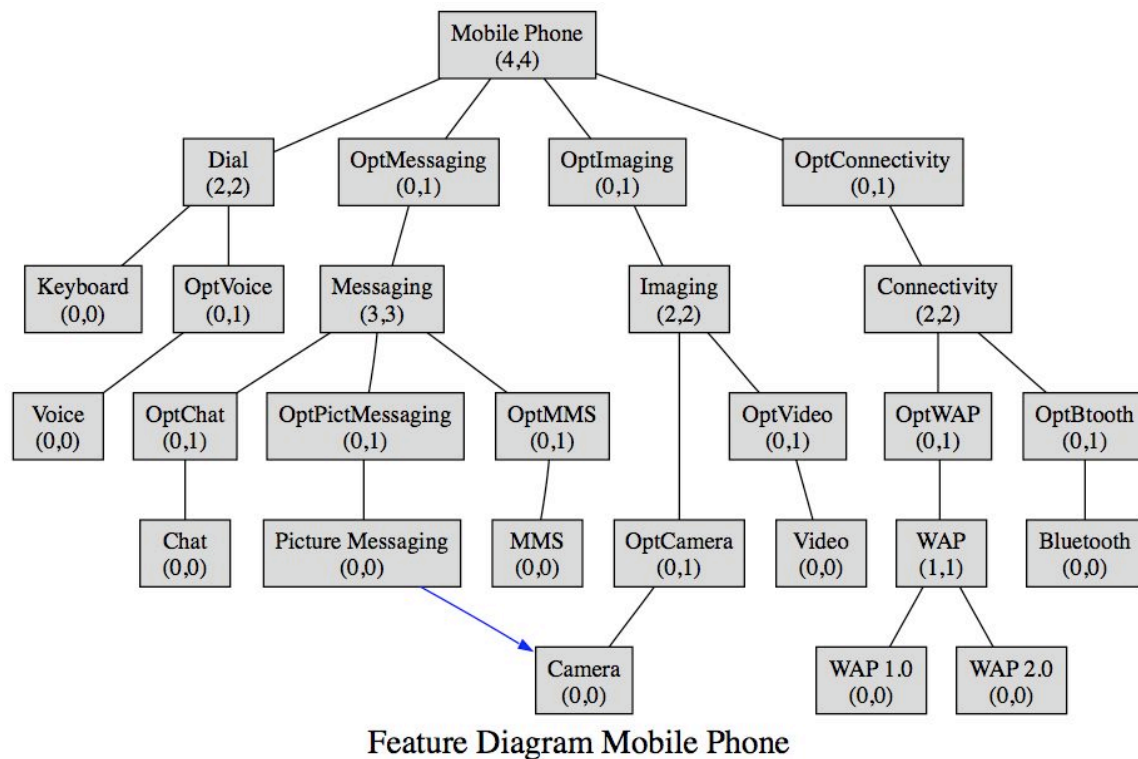
Figure 10.8: VFD Abstract Syntax: Mobile Phone PL

2. Develop Graphical Definition Model. This model contains information related to the graphical elements that will appear. Later, they will be connected to the constructs of language meta-model for which they provide representation and editing. This model defines the *concrete syntax* of the language.

3. Develop Tooling Definition Model. This model is used to design the palette and other menus, toolbars, etc., that will enable the user to manipulate the concepts of the language.

4. Develop Mapping Model. The mapping definition links the three previous models.

5. Develop Generator Model. This model enables to set the properties for code generation, similar to the familiar EMF Generator Model.

6. Generate Diagram Plug-in. Once the generator model is defined, a graphical editor Eclipse Plug-in is automatically generated for free.

At the end, a VFD graphical editor is provided. GMF allows to automate with minimum effort most of the tasks related to VFD editing and persistence. Finally, the generated tool has been customised and the graphical editor has been related to reasoning facilities. The final result is

an Eclipse Plug-in that associates the advantages of a SAT-solver and GMF to facilitate Feature Modelling (see Figure 10.9) and Reasoning (see Figure 10.10).
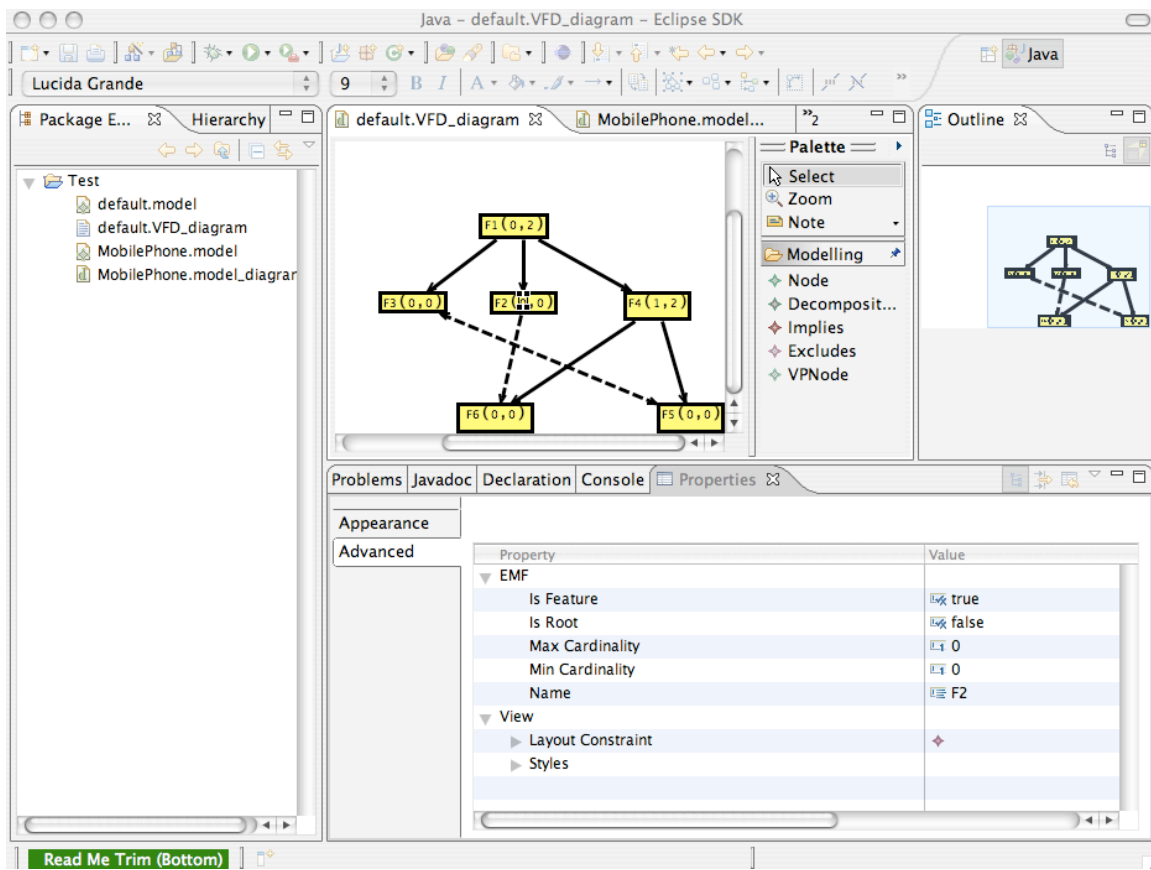
Figure 10.9: GMF Plugin: Feature Modelling
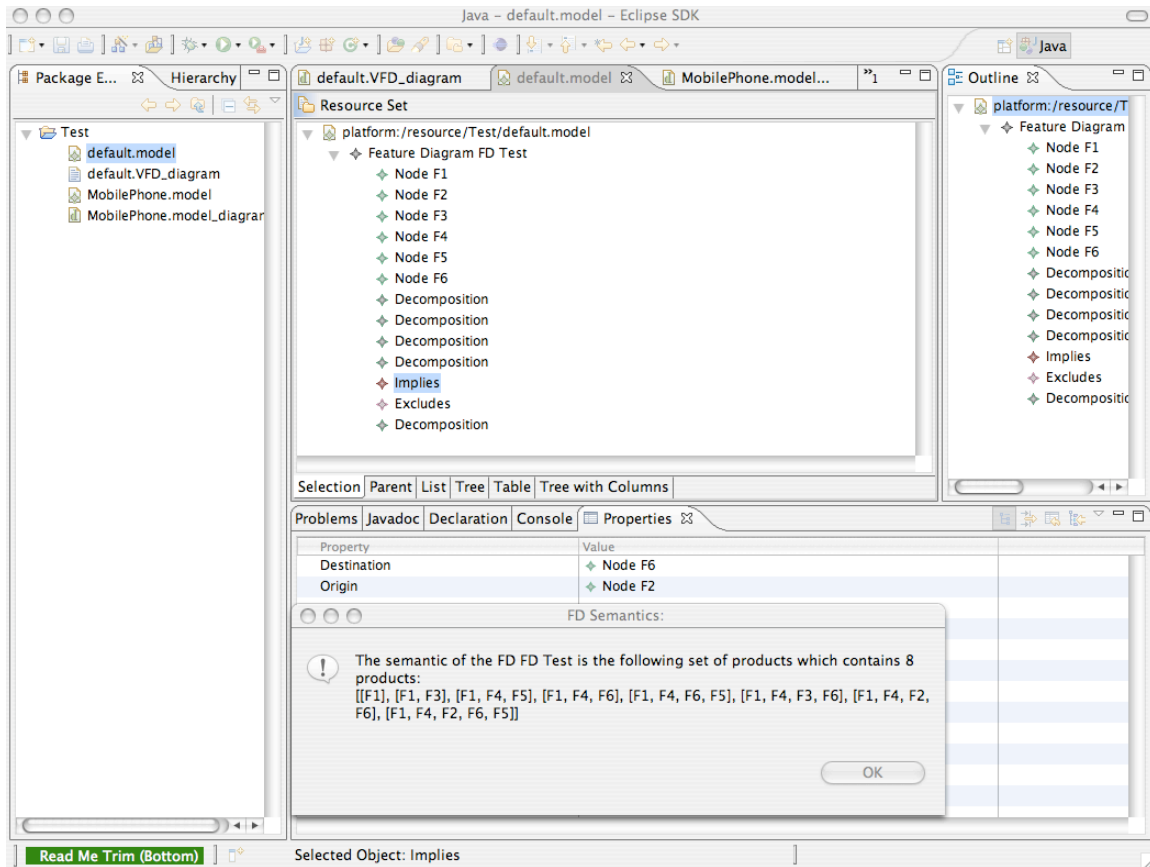
Figure 10.10: GMF Plugin: Feature Reasoning

10.6   Chapter Summary

In this chapter, we reported our implementation of a VFD reasoning tool mainly relying on a SAT-solver for its semantics and on GMF technologies for its graphical representation. We have described the expected functionalities of the tool and its basic architecture. Several crucial points have been underlined, especially the roles played by VFD within this tool and the encoding to translate *card*-nodes into BF in CNF. Finally, we have illustrated the tool with a simplified mobile phone example. For further details on the mobile phone example, please refer to Appendix I.

Part V

Conclusion and Future Work

Chapter 11

Conclusion and Future Work

F eature Diagrams (FDs) can be a precious help in mastering the complexity of variability management in the context of SPLE. They allow representing in a concise way the commonalities and the variabilities of a whole family of products in terms of their features. This thesis underlines that the current research on FDs is unfortunately fragmented and thus provides principles to remedy this situation. A formal approach is presented and designed to introduce more rigour in the motivation, definition and comparison of FD languages. Hence, examining their qualities will be more focused and productive. This quality analysis is necessary to avoid the proliferation of languages and constructs that are an additional source of misinterpretations and interoperability problems. In particular, the automation and formal underpinnings of FDs need more careful attention.

Our work contributes to the resolution of this situation by providing and applying a formal evaluation and comparison approach for FD languages. Throughout our investigation we have examined three central research questions for which we gave the following answers:

1. *RQ.1: Which qualities could be evaluated for FD languages?*

    This question has already been addressed from a broader perspective in the literature. Various language quality dimensions have been provided in SEQUAL, the global quality framework proposed in (Krogstie, 2001a). We used SEQUAL as a road-map to improve the quality of FD languages. The evaluation of language qualities is a complex endeavour involving many mutually dependent qualities. In addition, such qualities depend on the context of the use of the language. The importance of the studied qualities and criteria is mainly relative to the companies and projects concerned. FD languages are no exception.

2. *RQ.2: Which formal approaches facilitate evaluation and comparison of FD languages' quality, and how to use them?*

    This question involves the formal analysis of FD language quality. The main sub-questions to address are:

    - *Which quality dimensions are involved?*

Formal approaches do not aim at and are not able to deal with all the quality dimensions covered by SEQUAL. They are mainly interested in abstract syntax and semantics while issues concerning concrete syntax are not addressed. Hence, our approach does not consider concrete syntax and is mainly restricted to three SEQUAL quality dimensions: *Domain appropriateness*, *Comprehensibility appropriateness* and *Technical actor interpretation appropriateness*.

- *Which formal evaluation criteria can help to compare FD languages?*

  Defining FD languages formally puts us in a more comfortable position to compare and evaluate them. It allows us to:

  (a) specify decision problems related to FD languages, devise algorithms to solve these problems and assess their potential efficiency (*Computational Complexity*);

  (b) determine what the language is able to express (*Expressiveness*);

  (c) determine whether the diagrams written in one language are translatable into another language preserving the structure and the semantics of the original diagrams (*Embeddability*);

  (d) determine what is the worst possible impact on the size of the diagrams when there is a translation from one language to another (*Succinctness*).

  Computational Complexity, *Expressiveness, Embeddability* and *Succinctness* constitute our selection of criteria to evaluate and compare FD languages. The correspondence between these criteria and quality dimensions (from SEQUAL) they assess appears in Table 11.1.

	Domain appropriateness	Comprehensibility appropriateness	Technical actor interpretation appropriateness
Computational Complexity			√
Expressiveness	√		
Embeddability		√	√
Succinctness		√	√

Table 11.1: Language Qualities and Formal Criteria

- *How should such an analysis proceed?*

  Applying such criteria has a cost. On the one hand, the decision problems need to be formally specified, so that *Computational complexity* can be studied. On the other hand, to be compared according to *Expressiveness, Embeddability* and *Succinctness*, FD languages must be adapted to conform with two requirements: (1) they should be formal, meaning that they are given formally defined syntax, semantic domain and semantic

function and (2) they should share a common semantic domain.

The original contribution of this analysis is to suggest elegant solutions to facilitate the applicability of such criteria. Both informal and formal FD languages can then easily be made suitable for comparison.

 – The first solution provides automatically informal languages with a formal definition. We propose a configurable formal definition for a family of FD languages (FFD) and show how to configure it. FFD covers, but is not limited to, most of the existing FD languages.
 – The second solution allows us to compare formal languages for which semantic domains differ. We relate these semantic domains with abstraction functions. In addition, we study whether a translation between their respective syntactic domains exists that would ensure semantic equivalence between both languages. The abstraction function and syntax translation can thus be used to compare languages for expressiveness, embeddability and succinctness. Yet, since abstraction functions discard information from semantic domains, the relevance of this discarded information should still be studied carefully.

- *What are the main limitations to this analysis?*

The investigation reported here is not without limitation. The main limitations of our approach concern (a) its scope, (b) its perspective and (c) limitations associated to formal methods.

 (a) The main limitation of our work is explicit in its *scope*. The proposed method and its current results concern only formal language properties. In order not to over-interpret our conclusions, the reader should look at this work with a comprehensive view of model and language quality in mind. For example, with respect to the SEQUAL framework, in order to be accurate and effective, we have deliberately chosen to address only part of the qualities required from a "proper" modelling language. We are conscious that our criteria cover only parts of each of the three cited language qualities. In particular our formal approach only addresses qualities related to abstract syntax and formal semantics. Although concrete syntax is out of the thesis' scope, its importance should not be underestimated since it is extensively manipulated by language users and makes formality largely transparent for them.

 (b) We have only looked at language quality, adopting a *theoretical approach in an academic environment*. A complementary work would be to investigate models empirically. We emphasised the difficulty of such an endeavour because of the limited availability of "real" FDs. Nevertheless, we do not consider it impossible and can certainly learn a lot by observing how practitioners create and use FDs. Even, such kinds of observations can be found in (Hubaux et al., 2008). Although we have focused on studying theoretical properties of FD languages, we recognise that no formal semantics, nor criteria, can ever guarantee by itself that the languages help capture the right information (neither too little, nor too much) about the domain

being modelled. Only empirical research can help us give a convincing answer to this other aspect of domain appropriateness.

(c) *Formal methods* are extensively used in our approach. They bring many advantages particularly in terms of efficient and error-free tool support, independence from implementation, conciseness, unambiguity and completeness. Nevertheless, they are time-consuming, error-prone, difficult to validate and to comprehend for non-experts. In the context of FD languages, the advantages of formal methods are however very likely to outweigh the costs for several reasons: (1) the languages are relatively simple, (2) formality can be made largely transparent to the users (hidden behind a graphical concrete syntax), (3) the automation possibilities are many (Benavides et al., 2006; Schobbens et al., 2006, 2007), and (4) the information that FDs are used to convey is of critical importance for companies and therefore should suffer no ambiguity.

Finally, even within the clearly confined scope of our research, we face some *threats to validity*. Our examination of formal language properties was not supported by tools. All the formalisation of, and reasoning (comparisons, demonstrations of theorems) on languages were carried out by humans. Therefore, we cannot guarantee that human errors, miss- or over-interpretations are completely absent from our results. In addition, we need to draw the reader's attention to the fact that our formalisations were made only by considering published documents, without contacting the authors for clarifications nor testing their tools. Some of our formalisation choices might therefore only be due to the way things were phrased in the surveyed papers, or to an erroneous understanding from our part.

3. *RQ.3: How are existing FD languages evaluated according to this formal evaluation?*

A first set of results has been obtained from the application of this systematic method on a substantial part of the FD languages encountered in literature. The main conclusions are:

- Concerning *complexity analysis*, we have proved that most decision problems related to FD languages are non-trivial. Since no efficient algorithm exists to solve them, implementations are very likely to face tractability issues in some specific cases. We have shown that the expressively complete FD languages share the same complexity results:

    - the *Satisfiability* problem is *NP-Complete*;
    - the *Product-Checking* problem is *NP-Complete*;
    - the *Equivalence* problem is $coNP^{NP}$-*Complete*:
    - the *Inclusion* problem is $coNP^{NP}$-*Complete*;
    - the *Intersection* and *Union* problems are *Linear*.

Once a problem is known to be complex, its associated tractability issues could be minimised or at least circumscribed. For FDs, we follow Batory's suggestion (Batory, 2005) to use SAT-solvers (see Chapter 10). In addition, we mitigate each complexity result by

considering specific cases for which we determine a lower complexity. We also underline that sharing features or distinguishing primitive and non-primitive features may drastically increase complexity.

- Concerning *expressiveness analysis*, we have noticed that the studied FD languages are all expressively complete. In general, a FD language is expressively complete when it allows feature sharing. The simplest way to obtain feature sharing is to define FDs as DAGs. However, when FDs are limited to trees, constraint languages bring expressive completeness. Hence, the seminal FD language (OFT) was expressively complete and its extensions should provide explicit arguments based on other criteria than expressiveness. Since expressiveness is not sufficient to compare FD languages, we also study their embeddability and succinctness.

- Concerning *embeddability analysis*, we have noticed that FTs are not embeddable into FDs while the opposite is true. We have also observed that all FD languages are embeddable into EFD. EFD is even embeddable into itself (self-embeddable). This means that EFD is unnecessarily complex. Indeed, all constructs in EFD are easily definable using only one of them: *card*-nodes. These results justify the introduction of a new language called VFD that is minimal, non-redundant and still expressively complete. However, once languages share the same expressiveness and that no embedding can be found, succinctness is used as the finest criteria.

- Concerning *succinctness analysis*, we have determined five classes of succinctness and noticed that VFD is included in the lowest one. The five classes of succinctness are:

  (a) Propositional Logic;
  (b) BCs are exponentially-as succinct as Propositional Logic due to the sharing allowed in BC while not in Propositional Logic;
  (c) OFDs are exponentially-as succinct as BC due to the introduction of primitive and non-primitive features which implies that the corresponding BF should be existentially quantified;
  (d) RFDs are cubically-as succinct as OFD, due to the use of *or*-nodes;
  (e) EFDs are cubically-as succinct as RFD, due to the use of *card*-nodes.

  Embeddability and Succinctness analyses are highly interrelated. Indeed, languages often contain redundant constructs to improve their succinctness. For instance, adding specific constructs to represent *requires* and *excludes* constraints in OFD is redundant, however it clarifies diagrams.

- Concerning *semantic equivalence analysis* we have observed that

  - BC is semantically equivalent to FFD modulo an abstraction function. This abstraction function consists in discarding the non-primitive features from BC's semantic domain.
  - vDFD is not semantically equivalent to FFD although there exists an abstraction function between their semantic domains. The difference originates from the definition of their respective semantic functions. Indeed, vDFD follows an edge-based semantics while FFD follows a node-based semantics. However, we have been able

to obtain a semantic equivalence between FFD and vDFD by applying a preliminary transformation to the abstract syntax of FFD in order to behave as an edge-based semantics.

- We discover that two semantics were provided for BFT, namely BFT grammar and propositional semantics. However, they were not semantically equivalent. The difference originates from the definition of their semantic functions. Indeed, BFT propositional semantics generalised equivalence too much for *xor*-nodes. Finally, we determined that FFD and the BFT grammar semantics were semantically equivalent.

- In the end, our work suggests VFD as the language currently obtaining the best ranking according to the studied criteria. VFD is minimal and still expressively complete. VFD can embed all other studied FD languages and is included in the lowest class of succinctness.

11.1 Claimed Contributions

The contributions of this thesis are essentially methodological with a strong theoretical basis and substantial practical impacts.

11.1.1 Theoretical Contributions

A new semantics was defined and discussed for FFD. This definition is original in the sense that, unlike the majority of other proposals, it follows the principles provided in (Harel and Rumpe, 2004) and that it is configurable. In addition, FFD is formally defined and constitutes the theoretical basis of our analysis.

11.1.2 Methodological Contributions

The SEQUAL framework has served as a road-map and has been refined to improve the quality of FD languages. Accordingly, a method to evaluate both formal and informal FD language quality from a formal perspective was proposed. This method is based on three different aspects:

1. A set of principles was given to formally (re)define FD languages.

2. A set of formal criteria was proposed to partially assess specific quality dimensions of FD languages. These criteria are *expressiveness*, *embeddability*, *succinctness* and *complexity*.

3. A comparative semantic approach was proposed to render FD languages with different semantic domains comparable.

11.1.3 Practical Contributions

The aforementioned method was applied to compare and evaluate existing FD languages. The resulting contributions were to:

- provide a formal semantics to informal FD languages,

- compare, relate and discuss different proposals of semantics for FD languages including ours,

- determine classes of expressiveness, embeddability, succinctness and complexity of FD languages,

- fill these classes with the appropriate FD languages and

- compare and discuss FD languages and their constructs according to these classes and criteria.

In addition, a new rigorously defined and motivated FD language (VFD) was introduced. VFD has obtained the best scoring according to our selection of formal criteria. However, VFD was not considered as an end-user FD language. Hence, we did not provide VFD with an appropriate and explicit concrete syntax. The purpose of this language was to serve as a pivot language facilitating interoperability between FD languages. The translation from one FD language to another will thus be simplified and will guarantee that the original semantics is preserved.

Finally, a reasoning tool was implemented to support VFD.

11.2 Future Work

Ultimately, our research aims at accelerating the advent of a standard FD language of an overall excellent quality, including (1) unambiguous and appropriate syntax and semantics and (2) efficient and proved correct reference algorithms. Although the road ahead is still quite long, we are confident that the community can profit from our proposal. It could be used for example as part of an arsenal to elaborate a standard FD language. This standard would suffer no ambiguity, and its formal properties (among others) would be well known. In the end, it would enable reference algorithms to be devised that are efficient and already proven correct. To move forward in this direction, much work is still needed. The results should be validated (Section 11.2.1), extended (Section 11.2.2) and applied (Section 11.2.3). Additionally, the scope of the analysis can be broadened (Section 11.2.4).

11.2.1 Validating the Results

Having made explicit the semantics of several FD languages, we need to confront them with their proponents and, more generally, to the communities of researchers and practitioners working on the subject. Doing so, we will be able to correct possible misinterpretations (oversimplifications, arbitrary choices, etc.) we might have made, but also point out issues that were overlooked in informal definitions. We expect especially lively debates on *justification rule*, *edge-based vs. node-based semantics* (see discussion in Section 8.4.1) and on *primitive feature/node* (see discussion in Section 8.4.5).

In addition, the analysis of the tools implementing reasoning algorithms for FD languages would be a way to get a clearer understanding of their semantics.

11.2.2   Extending the Results

Our method has been applied to *informal* FD languages. A generic formalisation of all of them, FFD, was delivered and has helped gather precise results on them. Other informal languages could similarly benefit from formalisation, for instance Orthogonal Variability Modelling (OVM) (Pohl et al., 2005). In addition, some constructs found in the surveyed languages still have to be formalised, most notably, *layers*, *generalisation* and *implementation* links in FORM (Kang et al., 1998), *binding times* in (van Gurp et al., 2001) and *attributes* in (Benavides et al., 2005a).

Concerning *formal* FD languages, the semantic comparisons between FFD and vDFD and between FFD and BFT have helped gather precise results on them. More recently, several formalisation proposals (Czarnecki et al., 2005c,b; Benavides et al., 2005a; Wang et al., 2005a; Sun et al., 2005; Wang et al., 2005b; Asikainen et al., 2006; Janota and Kiniry, 2007) for FDs appeared in the literature. Each of them now needs to be carefully studied according to the proposed method and criteria. Furthermore, we claim that this comparison method could be used by the authors of any new proposals in order to justify it.

11.2.3   Applying the Results

Two main applications of our current results can be considered:

- One of the main expected outcomes of our work is the development of *efficient tool support* for FD languages. Several tools with reasoning capabilities already exist (Benavides et al., 2006) but, for most tasks, they have to face tough tractability issues. Our results (on formalisation and complexity, mainly) can help (1) verify the correctness of these algorithms, and (2) devise optimised algorithms.

- Various decision problems addressed in our work have been implemented. The next step is extending our tool with functionalities facilitating FD languages interoperability. VFD will play a central role here.

Of course, these applications will also be useful means to validate our results.

11.2.4   Extending the Scope

As mentioned before, the scope of our research is mainly limited to FD languages and a restricted number of qualities and criteria. In addition, gathering knowledge over FD languages should not be limited to definitions of FD languages found in the litterature. One alternative is to analyse FDs themselves and how practitioners draw them. Another alternative is to analyse tools that support FD languages and how they are used by practitioners. Accordingly, our formal approach should be (1) applied on other families of languages, (2) extended by other qualities and criteria and (3) complemented by an *empirical approach*.

1. Studying *other families of languages* may reveal interesting results. Our formal approach could also be transposed to cognate areas where existing modelling techniques face similar challenges. Particularly, we think of goal modelling techniques (van Lamsweerde, 2001; Mylopoulos, 2006), statecharts (Harel and Politi, 1998) or formalised subsets of UML (OMG, 2008).

2. Studying *other qualities and criteria* is equally important. In particular, issues related to concrete syntax are complementary to our current investigations. In our survey of FD languages, we observed that there were diverging views on this issue. Despite our focus on semantics, we do not underestimate the impact of a proper concrete syntax. In the end, this is the only thing that most language users will actually see. Evaluation and improvement of concrete syntax is an area of research that possesses an important body of knowledge which is currently being structured (Moody, 2006b,a), and of which FDs could take advantage. An important topic is to facilitate diagram scalability and to reduce the visual complexity of real-size models (Moody, 2006a).

3. Complementing our formal approach with an *empirical approach* could produce new results and could help validate our theoretical results. For instance, complexity results, which typically give our worst-case results, should be confronted with observations of the kinds (structure, size…) of models that are actually used by practitioners. If it turns out that most real FDs are trees (instead of DAGs), then our complexity results should not be considered too pessimistically. An empirical evaluation of the quality of their associated tools, following an approach similar to (Beuche et al., 2006), may also reveal complementary results.

Index

Bibliography

Akers, S. B., June 1978. Binary Decision Diagrams. IEEE Computer Transactions C- 27 (6), 509–516.

Anderson, M., 1997. "learning from diagrams". In: Special Issue on Diagrammatic Representation and Reasoning, Machine GRAPHICS & VISION. Vol. 6.

Antkiewicz, M., Czarnecki, K., 2004. FeaturePlugin: Feature Modeling plug-in for Eclipse. In: Proceedings of the Workshop on eclipse Technology eXchange (eTX'04). ACM Press, New York, NY, USA, pp. 67–72.

Apel, S., Lengauer, C., Batory, D., Möller, B., Kästner, C., 2007. An Algebra for Feature-Oriented Software Development. In: "MIP-0706". "Fakultät für Informatik und Mathematik, Universität Passau".

Armstrong, D. J., 2006. The Quarks of Object-Oriented Development. Communications of the ACM 49 (2), 123–128.

Asikainen, T., Männistö, T., Soininen, T., 2004. Representing Feature Models of Software Product Families Using a Configuration Ontology. ECAI 2004 Configuration Workshop.

Asikainen, T., Männistö, T., Soininen, T., 2006. A Unified Conceptual Foundation for Feature Modelling. In: Proceedings of the 10th International Software Product Line Conference. pp. 31–40.

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J., 2001. Component-based Product Line Engineering with UML. Component Software Series. Addison-Wesley.

AT&T, G., 2007.
  URL http://www.graphviz.org/,LastChecked:12/2007

Baas, L., Clements, P. C., Kazman, R., 2003. Software Architecture in Practice, Second Edition. SEI Series in Software Engineering. Addison-Wesley.

Bachmann, F., Bass, L., May 2001. Managing Variability in Software Architecture. In: Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'01). pp. 126–132.

Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesch, B., Vilbig, A., November 2003. A Meta-model for Representing Variability in Product Family Development. In: Verlag, S. (Ed.), Proceeding of the 5th International Workshop on Software Product-Family Engineering (PFE'03). Vol. LNCS 3014. Siena, Italy, pp. 66–80.

Batini, C., Ceri, S., Navathe, S. B., 1992. Conceptual Database Design: An Entity-Relationship Approach. Benjamin/Cummings.

Batory, D., 2003. A tutorial on feature oriented programming and product-lines. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 753–754.

Batory, D., October 22-26 2006. Feature Modularity for Product-Lines. Tutorial presented at (OOPSLA'06): Generative Programming and Component Engineering, Portland, Oregon, USA.

Batory, D., Benavides, D., Ruiz-Cortés, A., 2006. Automated Analysis of Feature Models: Challenges Ahead. Communications of the ACM December 06.

Batory, D. S., 2005. Feature Models, Grammars, and Propositional Formulas. In: Proceedings of the 9th International Conference on Software Product Lines (SPLC'05). pp. 7–20.

Becker, M., Feb. 2003. Towards a General Model of Variability in Product Families. In: Proceedings of the 1st Workshop on Software Variability Management (SVM'03). pp. 19–27.

Benavides, D., Ruiz-Cortés, A., Trinidad, P., 2005a. Automated Reasoning on Feature Models. Proceedings of the 17th International Conference (CAiSE'05) LNCS, Advanced Information Systems Engineering. 3520, 491–503.

Benavides, D., Ruiz-Cortés, A., Trinidad, P., Segura., S., 2006. A Survey on the Automated Analyses of Feature Models. In: Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06).

Benavides, D., Segura, S., Trinidad, P., Cortés, A. R., 2005b. Using Java CSP Solvers in the Automated Analyses of Feature Models. In: Post-proceedings Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05).

Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A., 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In: Proceedings of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'07). pp. 129–134.

Berenguer, G., Romero, R., Trujillo, J., Serrano, M., Piattini, M., 2005. A Set of Quality Indicators and Their Corresponding Metrics for Conceptual Models of Data Warehouses. In: Tjoa, A. M., Trujillo, J. (Eds.), DaWaK. Vol. 3589 of Lecture Notes in Computer Science. Springer, pp. 95–104.

Berre, S. D. L., Parrain, A., 2007.
URL http://www.sat4j.org/,LastChecked:12/2007

Bertolino, A., Fantechi, A., Gnesi, S., Lami, G., Maccari, A., September 2002. Use Case Description of Requirements for Product Lines. In: Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02). pp. 12–18.

Beuche, D., Birk, A., Dreier, H., Fleischmann, A., Galle, H., Heller, G., Janzen, D., John, I., Kolagari, R. T., von der Maßen, T., Wolfram, A., Dec. 2006. Report of the GI Work Group Requirements Management Tools for Product Line Engineering. Tech. Rep. AIB-2006-14, RWTH Aachen.

Biggerstaff, T. J., Perlis, A. J. (Eds.), 1989a. Software reusability: vol. 1, concepts and models. ACM, New York, NY, USA.

Biggerstaff, T. J., Perlis, A. J. (Eds.), 1989b. Software reusability: vol. 2, applications and experience. ACM, New York, NY, USA.

Boman, M., Janis A. Bubenko, J., Johannesson, P., Wangler, B., 1997. Conceptual modelling. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Bontemps, Y., Heymans, P., Schobbens, P.-Y., Trigaux, J.-C., August 2004. Semantics of FODA Feature Diagrams. In: Proceedings of the Workshop on Software Variability Management for Product Derivation: Towards Tool Support. Boston, pp. 48–58.

Bontemps, Y., Heymans, P., Schobbens, P.-Y., Trigaux, J.-C., 2005. Generic Semantics of Feature Diagrams Variants. In: Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05). IOS Press, pp. 58–77.

Boole, G., 1853. Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities. Dover classics of science and mathematics, New York.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley.

Bosch, J., Aug. 2002. Maturity and Evolution in Software Product Lines: Approaches, Artefacts, and Organization. In: Chastek, G. (Ed.), Proceedings of the Second Software Product Line Conference. LNCS 2379. Springer, San Diego, CA, pp. 257–271.

Bosch, J., September 2005. Keynote—Software Product Families in Nokia. Proceedings of the 9th International Conference on Software Product Lines (SPLC'05), 2–6.

Botterweck, G., O'Brien, L., Thiel, S., 2007. Model-driven derivation of product architectures. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, New York, NY, USA, pp. 469–472.

Bowen, J. P., Hinchey, M. G., 2006. Ten Commandments of Formal Methods ...Ten Years Later. Computer 39 (1), 40–48.

Brace, K. S., Rudell, R. L., Bryant, R. E., 1990. Efficient implementation of a bdd package. In: Proceedings of the 27th ACM/IEEE conference on Design automation (DAC'90). ACM Press, New York, NY, USA, pp. 40–45.

Brand, M. v. d., Deursen, A. v., Heering, J., Jong, H. d., Jonge, M. d., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J., 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. In: Wilhelm, R. (Ed.), Compiler Construction (CC'01). Vol. 2027 of Lecture Notes in Computer Science. Springer-Verlag, pp. 365–370.

Brownsword, L., Clements, P., 1996. A Case Study in Successful Product Line Development. Tech. Rep. CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University.

Cechticky, V., Pasetti1, A., Rohlik1, O., Schaufelberger, W., 2004. XML-Based Feature Modelling. In: Software Reuse: Methods, Techniques and Tools. pp. 101–114.

Chen, K., Zhang, W., Zhao, H., Mei, H., 2005. An Approach to Constructing Feature Models based on Requirements Clustering. In: Proceedings. 13th IEEE International Conference on Requirements Engineering. pp. 31–40.

Choi, Y.-H., Yoon, S., Shin, G.-S., Yang, Y., February 2006. An Extension of UML 2.0 for Representing Variability on Product Line Architecture. In: Proceedings of the IASTED International Conference on Software Engineering. Innsbruck, Austria, pp. 119–124.

Classen, A., Heymans, P., Schobbens, P.-Y., 2008. What's in a Feature: A Requirements Engineering Perspective. In: Fiadeiro, J. L., Inverardi, P. (Eds.), Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'08). Vol. 4961 of LNCS. Springer, pp. 16–30.

Classen, A., Tun, T., Heymans, P., Laney, R., Nuseibeh, B., 2007. On the Structure of Problem Variability: From Feature Diagrams to Problem Frames. In: Proceedings of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'07). Limerick, Ireland, pp. 109–117.

Clauss, M., September 2001. Generic Modeling using UML Extensions for Variability. In: OOPSLA 2001 Workshop on Domain Specific Visual Languages.

Clements, P., Northrop, L., 2008a. A Framework for Software Product Line Practice - Version 5.0 [online]. Carnegie Mellon, Software Engineering Institute URL: `http://www.sei.cmu.edu/productlines/framework.html` Last Checked: 12/2007, Pittsburgh, USA.

Clements, P., Northrop, L., 2008b. Product Line Hall of Fame. Carnegie Mellon, Software Engineering Institute URL: `http://www.sei.cmu.edu/productlines/plp_hof.html` Last Checked: 12/2007.

Clements, P. C., Northrop, L., Aug. 2001. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley.

Cohen, S., Sep. 2002. Product Line State of the Practice Report. Technical Note CMU/SEI-2002-TN-017, Software Engineering Institute, Carnegie Mellon University.

Cohen, S., Tekinerdogan, B., Czarnecki, K., 2002a. A case study on requirements specification: Driver Monitor. In: Proc. of the Workshop on Techniques for Exploiting Commonality Through Variability Management at the Second International Conference on Software Product Lines (SPLC'02).

Cohen, S., Tekinerdogan, B., Czarnecki, K., 2002b. Workshop on Techniques for Exploiting Commonality Through Variability Management (SPLC'02).

Cook, S. A., 1971. The complexity of theorem-proving procedures. In: STOC '71: Proceedings of the third annual ACM symposium on Theory of computing. ACM Press, New York, NY, USA, pp. 151–158.

Czarnecki, K., Antkiewicz, M., 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. R. (Eds.), GPCE. Vol. 3676 of Lecture Notes in Computer Science. Springer, pp. 422–437.

Czarnecki, K., Antkiewicz, M., Kim, C. H. P., Lau, S., Pietroszek, K., 2005a. fmp and fmp2rsm: Eclipse plug-ins for modeling features using model templates. In: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05). ACM Press, New York, NY, USA, pp. 200–201.

Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U. W., 2002. Generative Programming for Embedded Software: An Industrial Experience Report. In: Batory, D., Consel, C., Taha., W. (Eds.), ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering. Springer-Verlag, Berlin-Heidelberg, pp. 156–172.

Czarnecki, K., Helsen, S., Eisenecker, U., September 2004. Staged Configuration Using Feature Models. In: Proceedings of the Third International Conference on Software Product Lines (SPLC'O4). pp. 266–283.

Czarnecki, K., Helsen, S., Eisenecker, U., March 2005b. Formalizing Cardinality-based Feature Models and their Specialization. Software Process: Improvement and Practice 10 (1), 7–29.

Czarnecki, K., Helsen, S., Eisenecker, U., 2005c. Staged Configuration Using Feature Models. Software Process Improvement and Practice, Special Issue on Software Variability: Process and Management 10 (2), 143 – 169.

Dahl, O.-J., Nygaard, K., 1966. Simula - an algol-based simulation language. Commun. ACM 9 (9), 671–678.

de Boer, F. S., Palamidessi, C., 1994. Embedding as a tool for language comparison. Information and Computation 108 (1), 128–157.

Djebbi, O., Salinesi, C., 2006. Criteria for Comparing Requirements Variability Modeling Notations for Product Lines. Workshop on Comparative Evaluation in Requirements Engineering (CERE'06) 0, 20–35.

D'Souza, D. F., Wills, A. C., 1999. Objects, Components, and Frameworks with UML: the Catalysis Approach. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Eén, N., Sörensson, N., 2004. An Extensible SAT-solver. pp. 502–518.

Eisenecker, U. W., Czarnecki, K., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley.

Eriksson, M., Börstler, J., Borg, K., 2005. The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations. In: Proceedings of the 9th International Conference on Software Product Lines (SPLC'05). pp. 33–44.

Felleisen, M., 1990. On the Expressive Power of Programming Languages. In: Jones, N. D. (Ed.), Proceedings of the 3rd European Symposium on Programming (ESOP '90). Vol. 432 of LNCS. Springer Verlag, pp. 134–151.

Fey, D., Fajta, R., Boros, A., 2002a. Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. In: SPLC 2: Proceedings of the Second International Conference on Software Product Lines. Springer-Verlag, London, UK, pp. 198–216.

Fey, D., Fajta, R., Boros, A., Aug. 2002b. Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. In: Proceedings of the Second International Conference on Software Product Lines (SPLC'O2). pp. 198–216.

Forbus, K. D., de Kleer, J., 1993. Building problem solvers. MIT Press, Cambridge, MA, USA.

Foreman, J., April 23 1996. Product Line Based Software Development- Significant Results, Future Challenge. Software Technology Conference, Salt Lake City, UT, USA.

Genero, M., Jiménez, L., Piattini, M., October 2000. Measuring the Quality of Entity Relationship Diagrams. In: Laender, A. H. F., Liddle, S. W., Storey, V. C. (Eds.), Proceedings of the 19th International Conference on Conceptual Modeling (ER '00). Vol. LNCS 1920. Springer Berlin / Heidelberg, Salt Lake City, Utah, USA, pp. 513–526.

Genero, M., Piattini, M., Calero, C. (Eds.), 2005. Metrics for software conceptual models. Imperial College Press.

Gomaa, H., 2001. Modeling Software Product Lines with UML. In: Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications. pp. 27–31.

Gomaa, H., Shin, M. E., 2002. Multiple-View Meta-Modeling of Software Product Lines. In: Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems. IEEE Computer Society, Los Alamitos, CA, USA, p. 238.

Gomaa, H., Shin, M. E., July 2004. A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines. In: Proceedings of the 8th International Conference on Software Reuse: Methods, Techniques and Tools (ICSR '04). LNCS 3107. Springer, Madrid, Spain, pp. 274–285.

González-Baixauli, B., do Prado Leite, J. C. S., Mylopoulos, J., 2004. Visual Variability Analysis for Goal Models. In: RE. IEEE Computer Society, pp. 198–207.

Griss, M., Favaro, J., d'Alessandro, M., June 1998. Integrating Feature Modeling with the RSEB. In: Proceedings of the 5th International Conference on Software Reuse (ICSR'98). Vancouver, BC, Canada, pp. 76–85.

Haarslev, V., Möller, R., 2003. Racer: A core inference engine for the semantic web. In: Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON'03), located at the 2nd International Semantic Web Conference (ISWC'03), Sanibel Island, Florida, USA, October 20. pp. 27–36.

Halmans, G., Pohl, K., 2003. Communicating the Variability of a Software-product Family to Customers. Journal Software and Systems Modeling 2 (1), 15–36.

Harel, D., Politi, M., 1998. Modeling Reactive Systems with Statecharts: The Statemate Approach. McGraw-Hill, Inc., New York, NY, USA.

Harel, D., Rumpe, B., 2000. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Tech. Rep. MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science.

Harel, D., Rumpe, B., October 2004. Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer 37 (10), 64–72.

Henderson-Sellers, B., 1992. A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design and Implementation, a New Approach to Software Engineering. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Heymans, P., Schobbens, P.-Y., Trigaux, J.-C., Bontemps, Y., Matulevičius, R., Classen, A., June 2008. Evaluating Formal Properties of Feature Diagram Languages. IET special issue on Language Engineering. 2 (3), 281–302.

Hopcroft, J. E., Motwani, R., Ullman, J. D., November 2000. Introduction to Automata Theory, Languages, and Computation (2nd Edition). Addison Wesley.

Hubaux, A., Heymans, P., Benavides, D., 2008. Variability Modelling Challenges from the Trenches of an Open Source Product Line Re-Engineering Project. In: International Software Product Line Conference (SPLC'08).

IBM, G., Borland, 2007.
URL http://www.eclipse.org/gmf/,LastChecked:12/2007

IEEE Software Staff, 1994. Why Do So Many Reuse Programs Fail? IEEE Software 11 (5), 114–115.

Jackson, D., 2006. Software Abstractions: Logic, Language, and Analysis. The MIT Press.

Jackson, M., 1995. Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Jackson, M., 2001. Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Jacobson, I., Griss, M., Jonsson, P., 1997. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley.

Janota, M., Kiniry, J., 2007. Reasoning about Feature Models in Higher-Order Logic. Proceedings of the 11th International Conference on Software Product Lines (SPLC'O7) 0, 13–22.

Janssens, D., 1983. Node Label Controlled Graph Grammars. Tech. rep., University of Antwerpen.

Janssens, D., Rozenberg, G., 1980. On the structure of node label controlled graph languages. Inform. Sci. 20, 191–244.

John, I., Muthig, D., August 2002. Product Line Modeling with Generic Use Cases. In: Proceedings the second Software Product Lines Conference (SPLC '02).

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S., Nov. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals in Software Engineering 5, 143–168.

Kang, K. C., Lee, J., Donohoe, P., Jul. 2002. Feature-Oriented Product Line Engineering. IEEE Software 19 (4), 58–65.

Kiczales, G., 1996. Aspect-oriented programming. ACM Comput. Surv. 28 (4es), 154.

Kim, S.-K., Carrington, D., October 1999. Formalizing the UML Class Diagram Using Object-Z. In: In Proceedings of the Second International Conference on The Unified Modeling Language: Beyond the Standard (UML '99). LNSC 1723. Fort Collins, CO, USA, p. 753.

Kim, Y.-G., March, S. T., 1995. Comparing data modeling formalisms. Commun. ACM 38 (6), 103–115.

Kleene, S. C., 1952. Introduction to Metamathematics. Vol. 1 of Bibliotheca Mathematica. North-Holland, Amsterdam.

Kleppe, A., Warmer, J., Bast, W., April 2003. MDA Explained: The Model Driven Architecture–Practice and Promise. Addison-Wesley Professional.

Knauber, P., Muthig, D., Schmid, K., Widen, T., September 2000. Applying Product Line Concepts in Small and Medium-Sized Companies. IEEE SOFTWARE 17 (5), 88–95.

Kolberg, M., Magill, E., Wilson, M., November 2003. Compatibility Issues between Services supporting Networked Appliances. IEEE Communications 41 (11).

Krogstie, J., 1995. Conceptual Modeling for Computerized Information Systems Support in Organizations. Ph.D. thesis, Information Systems Group, Faculty of Electrical Engineering and Computer Science, The Norwegian Institute of Technology, The University of Trondheim.

Krogstie, J., 2001a. A Semiotic Approach to Quality in Requirements Specifications. In: Organizational Semiotics. pp. 231–249.

Krogstie, J., 2001b. Using a Semiotic Framework to Evaluate UML for the Development of Models of High Quality. Unified Modeling Language: System Analysis, Design and Develoment Issues, IDEA Group Publishing, 89–106.

Krogstie, J., Sindre, G., Jørgensen, H., 2006. Process Models Representing Knowledge for Action: a Revised Quality Framework. Eur. J. Inf. Syst. 15 (1), 91–102.

Krueger, C. W., 1992. Software reuse. ACM Comput. Surv. 24 (2), 131–183.

Krueger, C. W., April 2001. Using separation of concerns to simplify software product family engineering. In: Dagstuht Seminar No. 01161. Dagstuhl Castle, Wadern, Germany.

Kuusela, J., Savolainen, J., Jun. 2000. Requirements Engineering for Product Families. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE'00). ACM, Limerick, Ireland, pp. 61–69.

Lee, K., Kang, K. C., 2004. Feature Dependency Analysis for Product Line Component Design. In: ICSR. Vol. 3107 of Lecture Notes in Computer Science. Springer, pp. 69–85.

Lee, K., Kang, K. C., Kim, M., Park, S., 2006. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In: SPLC '06: Proceedings of the 10th International on Software Product Line Conference. IEEE Computer Society, Washington, DC, USA, pp. 103–112.

Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., Mylopoulos, J., September 2006. On Goal-based Variability Acquisition and Analysis. In: Proceedings of the 14th IEEE International Conference on Requirements Engineering (RE'06). IEEE Computer Society, Minneapolis, Minnesota.

Lindland, O. I., Sindre, G., Sølvberg, A., 1994. Understanding Quality in Conceptual Modeling. IEEE Software 11 (2), 42–49.

Maccari, A., September 2001. Industrial Keynote—Feature Modeling in an Industrial Context. In: Proceedings of the International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing (PLEES'01). Erfurt, Germany.

Maccari, A., Heie, A., 13-14 February 2003. Managing Infinite Variability. In: Proceedings of Software Variability Management Workshop (SVM'03). Groningen, The Nederlands, pp. 28–34.

Maier, R., 2001. Organizational Concepts and Measures for the Evaluation of Data Modeling. Developing quality complex database systems: practices, techniques and technologies, 1–27.

Mannion, M., Aug. 2002. Using First-Order Logic for Product Line Model Validation. In: Proceedings of the Second Software Product Line Conference (SPLC'02). LNCS 2379. Springer, San Diego, CA, pp. 176–187.

Matulevičius, R., Heymans, P., Opdahl, A. L., 2006. Comparison of Goal-oriented Languages using the UEML Approach. In: Proceedings of the 2nd International I-ESA 2006 Workshop on Enterprise Integration, Interoperability and Networking (EI2N'06).

McGuinness, D. L., van Harmelen, F., February 2004. OWL Web Ontology Language.
URL http://www.w3.org/TR/owl-features/,LastChecked:12/2007

McIlroy, D., 1968. Mass-Produced Software Components. In: Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany. pp. 88–98.

Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., Saval, G., October 2007. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In: Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07).

Mickel, A. B., Miner, J. F., Jensen, K., Wirth, N., 1991. Pascal user manual and report (4th ed.): ISO Pascal standard. Springer-Verlag New York, Inc., New York, NY, USA.

Moody, D. L., November 1998. Metrics for Evaluating the Quality of Entity Relationship Models. In: Ling, T. W., Ram, S., Lee, M.-L. (Eds.), Proceedings of the 17th International Conference on Conceptual Modeling (ER '98). Vol. 1507 of Lecture Notes in Computer Science. Springer, Singapore, pp. 211–225.

Moody, D. L., 2003. Measuring the Quality of Data Models: an Empirical Evaluation of the Use of Quality Metrics in Practice. In: Proceedings of the 11th European Conference on Information Systems (ECIS '03). Naples, Italy.

Moody, D. L., December 2005. Theoretical and Practical Issues in Evaluating the Quality of Conceptual Models: Current State and Future Directions. Data & Knowledge Engineering 55 (3), 243–276.

Moody, D. L., 2006a. Dealing with "Map Shock": A Systematic Approach for Managing Complexity in Requirements Modelling. In: Proceedings of the 12th Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ '06). Luxembourg.

Moody, D. L., 2006b. What Makes a Good Diagram? Improving the Cognitive Effectiveness of Diagrams in IS Development. In: Proceedings of the 15th International Conference in Information Systems Development (ISD'06).

Moody, D. L., Shanks, G. G., 1994. What Makes a Good Data Model? Evaluating the Quality of Entity Relationship Models. In: ER. Vol. 881 of LNCS. Springer, pp. 94–111.

Moody, D. L., Shanks, G. G., 2003. Improving the Quality of Data Models: Empirical Validation of a Quality Management Framework. Inf. Syst. 28 (6), 619–650.

Moody, S., 1991. Exploring Frameworks and Representations for Domain Specific Automatic Code Generation. In: Proceedings of the Fourth Workshop on Institutionalizing Software Reuse.

Mylopoulos, J., 2006. Goal-Oriented Requirements Engineering, Part II. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06). IEEE Computer Society, Washington, DC, USA, p. 4.

Northrop, L. M., July 2002. SEI's Software Product Line Tenets. IEEE Software 19 (4), 32–40.

OMG, 2008. UML 2.0 Superstructure Specification available from `http://www.omg.org/cgi-bin/doc?formal/05-07-04`, Last Checked: 01/2008.

Opdahl, A. L., Berio, G., 2006. Interoperable Language and Model Management using the UEML Approach. In: Proceedings of the 2006 international workshop on Global integrated model management (GaMMa '06). ACM Press, New York, NY, USA, pp. 35–42.

Owre, S., Rushby, J. M., , Shankar, N., jun 1992. PVS: A Prototype Verification System. In: Kapur, D. (Ed.), 11th International Conference on Automated Deduction (CADE'92). Vol. 607 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Saratoga, NY, pp. 748–752.

Papadimitriou, C. H., 1994. Computational Complexity. Addison-Wesley.

Parnas, D., Mar. 1976. On the Design and Development of Program Families. IEEE Transactions on Software Engineering SE-2 (1), 1–9.

Pohl, K., Bockle, G., van der Linden, F., July 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer.

Prasse, M., 1998. Evaluation of Object-Oriented Modelling Languages: A comparison between OML and UML. In: Schader, M., Korthaus, A. (Eds.), The Unified Modeling Language – Technical Aspects and Applications. Physica-Verlag, Heidelberg, pp. 58–75.

Pure-systems, July 2007. pure::variants Tool URL:`http://www.software-reuse.org/purevariants/`, Last Checked: 12/2007.

Reingruber, M. C., Gregory, W. W., 1994. The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models. John Wiley & Sons, Inc., New York, NY, USA.

Riebisch, M., 2003. Towards a More Precise Definition of Feature Models. Position Paper In Modelling Variability for Object-Oriented Product Lines.

Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I., Jun. 2002. Extending Feature Diagrams with UML Multiplicities. In: Proceedings of the 6th Conference on Integrated Design and Process Technology (IDPT '02). Pasadena, CA.

Rohlik, O., Pasetti, A., Oct. 2006. Prototype Family Modelling Tool. Tech. rep., Swiss Federal Institute of Technology Zurich.

Rossi, M., Brinkkemper, S., 1996. Complexity Metrics for Systems Development Methods and Techniques. Inf. Syst. 21 (2), 209–227.

Ryan, M., Schobbens, P. Y., 2004. FireWorks: A Formal Transformation-Based Model-Driven Approach to Features in Product Lines. In: Proceedings of the Worshop on Software Variability Management for Product Derivation - Towards Tool Support.

Saaltink, M., 1997. The Z/EVES System. In: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation (ZUM '97). Springer-Verlag, London, UK, pp. 72–85.

Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y., September 2006. Feature Diagrams: A Survey and A Formal Semantics. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06). Minneapolis, Minnesota, USA, pp. 139–148.

Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y., February 2007. Generic semantics of feature diagrams. Computer Networks (2007) special issue on feature interactions in emerging application domains 51, 456–479.

Schuette, R., Rotthowe, T., 1998. The Guidelines of Modeling - An Approach to Enhance the Quality in Information Models. In: Proceedings of the 17th International Conference on Conceptual Modeling (ER '98). Springer-Verlag, London, UK, pp. 240–254.

Sebesta, R. W., 1993. Concepts of programming languages, 2nd Edition. Benjamin/Cummings Publishing Company,Inc.

Shannon, C., 1937. A Symbolic Analysis of Relay and Switching Circuits. Master's thesis, MIT.

Shannon, C., 1938. A Symbolic Analysis of Relay and Switching Circuits. Transactions American Institute of Electrical Engineers 57, 713–723.

Sheffer, H. M., 1913. A set of five independent postulates for Boolean algebras, with application to logical constants. Trans. AMS 14, 481–488.

Shoham, Y., 1990. Agent-Oriented Programming. Tech. Rep. STAN-CS-90-1335, Stanford University Computer Science Department.

Simos, M., Creps, D., Klinger, C., Levine, L., Allemang, D., 1996. Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling (ODM) Guidebook Version 2.0 (STARS-VC-A025/001/00). Tech. rep., Manassas, VA: Lockheed Martin Tactical Defense Systems.

Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J., Sep. 2004. COVAMOF: A Framework for Modeling Variability in Software Product Families. In: Proceedings of the Third Software Product Line Conference (SPLC'04). Boston, MA, USA, pp. 197–213.

Sinz, C., Oct. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05). Sitges, Spain, pp. 827–831.

Soininen, T., Niemelä, I., Tiihonen, J., Sulonen, R., March 2001. Representing configuration knowledge with weight constraint rules. In: Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming. AAAI Press, Stanford, USA, pp. 195–201.

Spivey, J. M., 1988. Understanding Z: a specification language and its formal semantics. Cambridge University Press, New York, NY, USA.

Stoy, J. E., 1977. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA.

Streitferdt, D., Riebisch, M., Philippow, I., April 2003. Formal Details of Relations in Feature Models. In: Proceedings of the 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03). IEEE Computer Society Press, Huntsville, Alabama, USA, pp. 297–304.

Sun, J., Zhang, H., Li, Y. F., Wang, H., 2005. Formal Semantics and Verification for Feature Modeling. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '05). pp. 303–312.

Sutherland, W. R., 1966. The On-Line Graphical Specification of Computer Procedures. Tech. rep., Ph.D Thesis - Massachusetts Institute of Technology. Dept. of Electrical Engineering.

Svahnberg, M., Bosch, J., Mar. 2000. Issues Concerning Variability in Software Product Lines. In: Linden, F. v. d. (Ed.), In Proceedings of the 3th International Worshop on Software Architectures for Product Lines (IWSAPF '00). LNCS 1951. Springer, Las Palmas de Gran Canaria, Spain, pp. 146–157.

Svahnberg, M., van Gurp, J., Bosch, J., 2002. A Taxonomy of Variability Realization Techniques. Tech. rep., Blekinge Institute of Technology, Sweden.

Szlenk, M., 2006. Formal Semantics and Reasoning about UML Class Diagram. In: Proceedings of the International Conference on Dependability of Computer Systems (DepCos-RELCOMEX '06). Vol. 0. IEEE Computer Society, Los Alamitos, CA, USA, pp. 51–59.

Tarski, A., 1956. Logics, Semantics and Metamathematics. Clarendon Press.

Thiel, S., Ferber, S., Fischer, T., Hein, A., Schlick, M., Mar. 2001. A Case Study in Applying the Product Line Approach for Car Periphery Supervision Systems. In: Proceedings of In-Vehicle Software. Detroit, Michigan, USA, pp. 43–55.

Thomas, D., 2004. MDA: Revenge of the Modelers or UML Utopia? IEEE Software 21 (3), 15–17.

Thomas, D., Barry, B. M., 2003. Model Driven Development: the Case for Domain Oriented Programming. In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented

programming, systems, languages, and applications (OOPSLA '03). ACM Press, New York, NY, USA, pp. 2–7.

Trigaux, J.-C., Heymans, P., September 2005. Uniform Variability Management at the Model Level. In: Proceedings of 1st European Workshop on Model Transformation (EWMT'05). Rennes, France.

Trigaux, J.-C., Heymans, P., Schobbens, P.-Y., Classen, A., September 2006. Comparative semantics of Feature Diagrams : FFD vs vDFD. In: Proceedings of Workshop on Comparative Evaluation in Requirements Engineering (CERE'06). Minneapolis, Minnesota, USA.

Tsang, E., 1995. Foundations of Constraint Satisfaction. Academic Press.

van der Maßen, T., Lichter, H., September 2002. Modeling Variability by UML Use Case Diagrams. In: Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02). AVAYA labs, pp. 19–25.

van Deursen, A., Klint, P., 2002. Domain-Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology 10 (1), 1–17.

van Griethuysen, J. J., January 1982. Concepts and Terminology for the Conceptual Schema and the Information Base. International Organization for Standardization.

van Gurp, J., Bosch, J., Svahnberg, M., 2001. On the Notion of Variability in Software Product Lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01).

van Lamsweerde, A., 2001. Goal-Oriented Requirements Engineering: a Guided Tour. In: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE'01). pp. 249–262.

van Leeuwen, J. (Ed.), 1990. Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. Elsevier and MIT Press.

Vollmer, H., 1999. Introduction to Circuit Complexity: A Uniform Approach. Springer-Verlag, New York.

Walnes, X. J., Schaible, J., 2007.
     URL `http://xstream.codehaus.org/,LastChecked:12/2007`

Wang, H., Fang, L. Y., Sun, J., Zhang, H., Pan, J. Z., 2005a. A Semantic Web Approach to Feature Modeling and Verification. In: Proceedings of the International Workshop on Semantic Web Enabled Software Engineering (SWESE'05).

Wang, H., Li, Y. F., Sun, J., Zhang, H., 2005b. Verify Feature Models using protegeOWL. In: Special interest tracks and posters of the 14th international conference on World Wide Web (WWW '05). ACM Press, New York, NY, USA, pp. 1038–1039.

Weiss, D. M., Lai, C. T. R., 1999. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley.

Woodcock, J. C. P., 1993. Using Standard Z. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, in preparation.

Wooldridge, M., Jennings, N. R., 1995. Intelligent Agents: Theory and Practice. Knowledge Engineering Review 10 (2), 115–152.

Ziadi, T., Hélouët, L., Jézéquel, J.-M., Sep. 2002. Modelling Behaviors in Product Lines. In: Proceedings of the International Workshop on Requirements Engineering for Product Lines (REPL'02). pp. 33–38.

Ziadi, T., Hélouët, L., Jézéquel, J.-M., November 2003. Towards a UML Profile for Software Product Lines. Proceedings of the Fifth International Workshop on Product Family Engineering (PFE'03).

Appendix I: Mobile Phone Systems Example

This appendix gathers the main results concerning our illustrative example. First, in Section 11.2.4 we recall the concrete syntax of the OFD representing a simplified PL of mobile phone systems. Then, in Section 11.2.4, we translate this OFD into VFD and present graphically its abstract syntax. Afterwards, the FD semantics is provided in Section 11.2.4. Finally, we present the results of the translations of VFD into three different formats: CNF (Section 11.2.4), DOT (Section 11.2.4) and XML (Section 11.2.4).

Mobile Phone Concrete Syntax
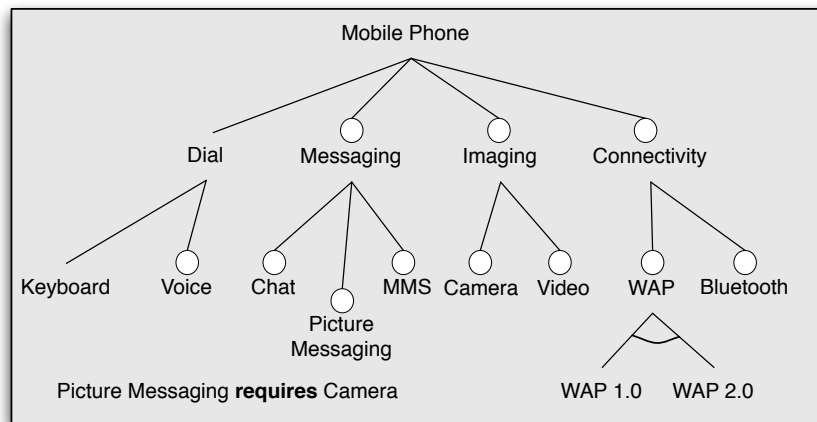
Figure 11.1: FORM FD: Mobile Phone PL
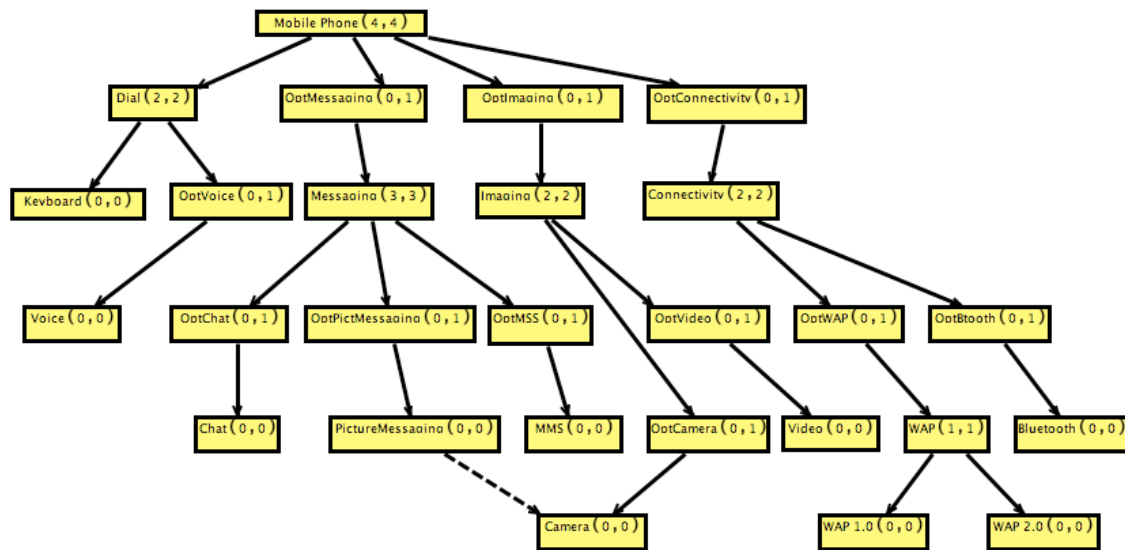
Figure 11.2: VFD Abstract Syntax: Mobile Phone PL

Mobile Phone Abstract Syntax

Mobile Phone Semantics

When the set of primitive features is limited to leaf features (Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 1.0, WAP 2.0) the FD (Figure 11.1) represents 288 valid different products. For each product the number of included primitive features is given between parenthesis and the set of primitive features between brackets. The set of products is sorted according to the number of primitive features included in a valid product.

    [Product (1): [Keyboard]
, Product (2): [Keyboard, Voice]
, Product (2): [Keyboard, Bluetooth]
, Product (2): [Keyboard, WAP 2.0]
, Product (2): [Keyboard, WAP 1.0]
, Product (2): [Keyboard, Video]
, Product (2): [Keyboard, Camera]
, Product (2): [Keyboard, MMS]
, Product (2): [Keyboard, Chat]
, Product (3): [Keyboard, Bluetooth, WAP 2.0]
, Product (3): [Keyboard, Bluetooth, WAP 1.0]
, Product (3): [Keyboard, Voice, Bluetooth]
, Product (3): [Keyboard, Voice, WAP 2.0]
, Product (3): [Keyboard, Voice, WAP 1.0]

, Product (3): [Keyboard, Camera, Video]
, Product (3): [Keyboard, Voice, Video]
, Product (3): [Keyboard, Voice, Camera]
, Product (3): [Keyboard, Video, Bluetooth]
, Product (3): [Keyboard, Video, WAP 2.0]
, Product (3): [Keyboard, Video, WAP 1.0]
, Product (3): [Keyboard, Camera, Bluetooth]
, Product (3): [Keyboard, Camera, WAP 2.0]
, Product (3): [Keyboard, Camera, WAP 1.0]
, Product (3): [Keyboard, Chat, MMS]
, Product (3): [Keyboard, Voice, MMS]
, Product (3): [Keyboard, Voice, Chat]
, Product (3): [Keyboard, MMS, Bluetooth]
, Product (3): [Keyboard, MMS, WAP 2.0]
, Product (3): [Keyboard, MMS, WAP 1.0]
, Product (3): [Keyboard, Chat, Bluetooth]
, Product (3): [Keyboard, Chat, WAP 2.0]
, Product (3): [Keyboard, Chat, WAP 1.0]
, Product (3): [Keyboard, MMS, Video]
, Product (3): [Keyboard, MMS, Camera]
, Product (3): [Keyboard, Picture Messaging, Camera]
, Product (3): [Keyboard, Chat, Video]
, Product (3): [Keyboard, Chat, Camera]
, Product (4): [Keyboard, Voice, Bluetooth, WAP 1.0]
, Product (4): [Keyboard, Voice, Bluetooth, WAP 2.0]
, Product (4): [Keyboard, Voice, Camera, Video]
, Product (4): [Keyboard, Video, Bluetooth, WAP 1.0]
, Product (4): [Keyboard, Video, Bluetooth, WAP 2.0]
, Product (4): [Keyboard, Camera, Bluetooth, WAP 2.0]
, Product (4): [Keyboard, Camera, Bluetooth, WAP 1.0]
, Product (4): [Keyboard, Camera, Video, Bluetooth]
, Product (4): [Keyboard, Camera, Video, WAP 2.0]
, Product (4): [Keyboard, Camera, Video, WAP 1.0]
, Product (4): [Keyboard, Voice, Video, Bluetooth]
, Product (4): [Keyboard, Voice, Video, WAP 2.0]
, Product (4): [Keyboard, Voice, Video, WAP 1.0]
, Product (4): [Keyboard, Voice, Camera, Bluetooth]
, Product (4): [Keyboard, Voice, Camera, WAP 2.0]
, Product (4): [Keyboard, Voice, Camera, WAP 1.0]
, Product (4): [Keyboard, Voice, Chat, MMS]
, Product (4): [Keyboard, MMS, Bluetooth, WAP 1.0]
, Product (4): [Keyboard, MMS, Bluetooth, WAP 2.0]
, Product (4): [Keyboard, Chat, Bluetooth, WAP 2.0]

, Product (4): [Keyboard, Chat, Bluetooth, WAP 1.0]
, Product (4): [Keyboard, Chat, MMS, Bluetooth]
, Product (4): [Keyboard, Chat, MMS, WAP 2.0]
, Product (4): [Keyboard, Chat, MMS, WAP 1.0]
, Product (4): [Keyboard, Voice, MMS, Bluetooth]
, Product (4): [Keyboard, Voice, MMS, WAP 2.0]
, Product (4): [Keyboard, Voice, MMS, WAP 1.0]
, Product (4): [Keyboard, Voice, Chat, Bluetooth]
, Product (4): [Keyboard, Voice, Chat, WAP 2.0]
, Product (4): [Keyboard, Voice, Chat, WAP 1.0]
, Product (4): [Keyboard, MMS, Camera, Video]
, Product (4): [Keyboard, Picture Messaging, Camera, Video]
, Product (4): [Keyboard, Picture Messaging, MMS, Camera]
, Product (4): [Keyboard, Chat, Camera, Video]
, Product (4): [Keyboard, Chat, MMS, Video]
, Product (4): [Keyboard, Chat, MMS, Camera]
, Product (4): [Keyboard, Chat, Picture Messaging, Camera]
, Product (4): [Keyboard, Voice, MMS, Video]
, Product (4): [Keyboard, Voice, MMS, Camera]
, Product (4): [Keyboard, Voice, Picture Messaging, Camera]
, Product (4): [Keyboard, Voice, Chat, Video]
, Product (4): [Keyboard, Voice, Chat, Camera]
, Product (4): [Keyboard, MMS, Video, Bluetooth]
, Product (4): [Keyboard, MMS, Video, WAP 2.0]
, Product (4): [Keyboard, MMS, Video, WAP 1.0]
, Product (4): [Keyboard, MMS, Camera, Bluetooth]
, Product (4): [Keyboard, MMS, Camera, WAP 2.0]
, Product (4): [Keyboard, MMS, Camera, WAP 1.0]
, Product (4): [Keyboard, Picture Messaging, Camera, Bluetooth]
, Product (4): [Keyboard, Picture Messaging, Camera, WAP 2.0]
, Product (4): [Keyboard, Picture Messaging, Camera, WAP 1.0]
, Product (4): [Keyboard, Chat, Video, Bluetooth]
, Product (4): [Keyboard, Chat, Video, WAP 2.0]
, Product (4): [Keyboard, Chat, Video, WAP 1.0]
, Product (4): [Keyboard, Chat, Camera, Bluetooth]
, Product (4): [Keyboard, Chat, Camera, WAP 2.0]
, Product (4): [Keyboard, Chat, Camera, WAP 1.0]
, Product (5): [Keyboard, Camera, Video, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Camera, Video, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Voice, Video, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Voice, Video, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Voice, Camera, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Voice, Camera, Bluetooth, WAP 1.0]

, Product (5): [Keyboard, Voice, Camera, Video, Bluetooth]
, Product (5): [Keyboard, Voice, Camera, Video, WAP 2.0]
, Product (5): [Keyboard, Voice, Camera, Video, WAP 1.0]
, Product (5): [Keyboard, Chat, MMS, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Chat, MMS, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Voice, MMS, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Voice, MMS, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Voice, Chat, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Voice, Chat, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Voice, Chat, MMS, Bluetooth]
, Product (5): [Keyboard, Voice, Chat, MMS, WAP 2.0]
, Product (5): [Keyboard, Voice, Chat, MMS, WAP 1.0]
, Product (5): [Keyboard, Picture Messaging, MMS, Camera, Video]
, Product (5): [Keyboard, Chat, MMS, Camera, Video]
, Product (5): [Keyboard, Chat, Picture Messaging, Camera, Video]
, Product (5): [Keyboard, Chat, Picture Messaging, MMS, Camera]
, Product (5): [Keyboard, Voice, MMS, Camera, Video]
, Product (5): [Keyboard, Voice, Picture Messaging, Camera, Video]
, Product (5): [Keyboard, Voice, Picture Messaging, MMS, Camera]
, Product (5): [Keyboard, Voice, Chat, Camera, Video]
, Product (5): [Keyboard, Voice, Chat, MMS, Video]
, Product (5): [Keyboard, Voice, Chat, MMS, Camera]
, Product (5): [Keyboard, Voice, Chat, Picture Messaging, Camera]
, Product (5): [Keyboard, MMS, Video, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, MMS, Video, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, MMS, Camera, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, MMS, Camera, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, MMS, Camera, Video, Bluetooth]
, Product (5): [Keyboard, MMS, Camera, Video, WAP 2.0]
, Product (5): [Keyboard, MMS, Camera, Video, WAP 1.0]
, Product (5): [Keyboard, Picture Messaging, Camera, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Picture Messaging, Camera, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Picture Messaging, Camera, Video, Bluetooth]
, Product (5): [Keyboard, Picture Messaging, Camera, Video, WAP 2.0]
, Product (5): [Keyboard, Picture Messaging, Camera, Video, WAP 1.0]
, Product (5): [Keyboard, Picture Messaging, MMS, Camera, Bluetooth]
, Product (5): [Keyboard, Picture Messaging, MMS, Camera, WAP 2.0]
, Product (5): [Keyboard, Picture Messaging, MMS, Camera, WAP 1.0]
, Product (5): [Keyboard, Chat, Video, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Chat, Video, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Chat, Camera, Bluetooth, WAP 2.0]
, Product (5): [Keyboard, Chat, Camera, Bluetooth, WAP 1.0]
, Product (5): [Keyboard, Chat, Camera, Video, Bluetooth]

, Product (5): [Keyboard, Chat, Camera, Video, WAP 2.0]
, Product (5): [Keyboard, Chat, Camera, Video, WAP 1.0]
, Product (5): [Keyboard, Chat, MMS, Video, Bluetooth]
, Product (5): [Keyboard, Chat, MMS, Video, WAP 2.0]
, Product (5): [Keyboard, Chat, MMS, Video, WAP 1.0]
, Product (5): [Keyboard, Chat, MMS, Camera, Bluetooth]
, Product (5): [Keyboard, Chat, MMS, Camera, WAP 2.0]
, Product (5): [Keyboard, Chat, MMS, Camera, WAP 1.0]
, Product (5): [Keyboard, Chat, Picture Messaging, Camera, Bluetooth]
, Product (5): [Keyboard, Chat, Picture Messaging, Camera, WAP 2.0]
, Product (5): [Keyboard, Chat, Picture Messaging, Camera, WAP 1.0]
, Product (5): [Keyboard, Voice, MMS, Video, Bluetooth]
, Product (5): [Keyboard, Voice, MMS, Video, WAP 2.0]
, Product (5): [Keyboard, Voice, MMS, Video, WAP 1.0]
, Product (5): [Keyboard, Voice, MMS, Camera, Bluetooth]
, Product (5): [Keyboard, Voice, MMS, Camera, WAP 2.0]
, Product (5): [Keyboard, Voice, MMS, Camera, WAP 1.0]
, Product (5): [Keyboard, Voice, Picture Messaging, Camera, Bluetooth]
, Product (5): [Keyboard, Voice, Picture Messaging, Camera, WAP 2.0]
, Product (5): [Keyboard, Voice, Picture Messaging, Camera, WAP 1.0]
, Product (5): [Keyboard, Voice, Chat, Video, Bluetooth]
, Product (5): [Keyboard, Voice, Chat, Video, WAP 2.0]
, Product (5): [Keyboard, Voice, Chat, Video, WAP 1.0]
, Product (5): [Keyboard, Voice, Chat, Camera, Bluetooth]
, Product (5): [Keyboard, Voice, Chat, Camera, WAP 2.0]
, Product (5): [Keyboard, Voice, Chat, Camera, WAP 1.0]
, Product (6): [Keyboard, Voice, Camera, Video, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Voice, Camera, Video, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Voice, Chat, MMS, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Voice, Chat, MMS, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Chat, Picture Messaging, MMS, Camera, Video]
, Product (6): [Keyboard, Voice, Picture Messaging, MMS, Camera, Video]
, Product (6): [Keyboard, Voice, Chat, MMS, Camera, Video]
, Product (6): [Keyboard, Voice, Chat, Picture Messaging, Camera, Video]
, Product (6): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera]
, Product (6): [Keyboard, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, MMS, Camera, Video, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Picture Messaging, Camera, Video, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Picture Messaging, Camera, Video, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Picture Messaging, MMS, Camera, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Picture Messaging, MMS, Camera, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Picture Messaging, MMS, Camera, Video, Bluetooth]
, Product (6): [Keyboard, Picture Messaging, MMS, Camera, Video, WAP 2.0]

, Product (6): [Keyboard, Picture Messaging, MMS, Camera, Video, WAP 1.0]
, Product (6): [Keyboard, Chat, Camera, Video, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Chat, Camera, Video, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Chat, MMS, Video, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Chat, MMS, Video, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Chat, MMS, Camera, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Chat, MMS, Camera, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Chat, MMS, Camera, Video, Bluetooth]
, Product (6): [Keyboard, Chat, MMS, Camera, Video, WAP 2.0]
, Product (6): [Keyboard, Chat, MMS, Camera, Video, WAP 1.0]
, Product (6): [Keyboard, Chat, Picture Messaging, Camera, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Chat, Picture Messaging, Camera, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Chat, Picture Messaging, Camera, Video, Bluetooth]
, Product (6): [Keyboard, Chat, Picture Messaging, Camera, Video, WAP 2.0]
, Product (6): [Keyboard, Chat, Picture Messaging, Camera, Video, WAP 1.0]
, Product (6): [Keyboard, Chat, Picture Messaging, MMS, Camera, Bluetooth]
, Product (6): [Keyboard, Chat, Picture Messaging, MMS, Camera, WAP 2.0]
, Product (6): [Keyboard, Chat, Picture Messaging, MMS, Camera, WAP 1.0]
, Product (6): [Keyboard, Voice, MMS, Video, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Voice, MMS, Video, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Voice, MMS, Camera, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Voice, MMS, Camera, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Voice, MMS, Camera, Video, Bluetooth]
, Product (6): [Keyboard, Voice, MMS, Camera, Video, WAP 2.0]
, Product (6): [Keyboard, Voice, MMS, Camera, Video, WAP 1.0]
, Product (6): [Keyboard, Voice, Picture Messaging, Camera, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Voice, Picture Messaging, Camera, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Voice, Picture Messaging, Camera, Video, Bluetooth]
, Product (6): [Keyboard, Voice, Picture Messaging, Camera, Video, WAP 2.0]
, Product (6): [Keyboard, Voice, Picture Messaging, Camera, Video, WAP 1.0]
, Product (6): [Keyboard, Voice, Picture Messaging, MMS, Camera, Bluetooth]
, Product (6): [Keyboard, Voice, Picture Messaging, MMS, Camera, WAP 2.0]
, Product (6): [Keyboard, Voice, Picture Messaging, MMS, Camera, WAP 1.0]
, Product (6): [Keyboard, Voice, Chat, Video, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Voice, Chat, Video, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Voice, Chat, Camera, Bluetooth, WAP 2.0]
, Product (6): [Keyboard, Voice, Chat, Camera, Bluetooth, WAP 1.0]
, Product (6): [Keyboard, Voice, Chat, Camera, Video, Bluetooth]
, Product (6): [Keyboard, Voice, Chat, Camera, Video, WAP 2.0]
, Product (6): [Keyboard, Voice, Chat, Camera, Video, WAP 1.0]
, Product (6): [Keyboard, Voice, Chat, MMS, Video, Bluetooth]
, Product (6): [Keyboard, Voice, Chat, MMS, Video, WAP 2.0]
, Product (6): [Keyboard, Voice, Chat, MMS, Video, WAP 1.0]

, Product (6): [Keyboard, Voice, Chat, MMS, Camera, Bluetooth]
, Product (6): [Keyboard, Voice, Chat, MMS, Camera, WAP 2.0]
, Product (6): [Keyboard, Voice, Chat, MMS, Camera, WAP 1.0]
, Product (6): [Keyboard, Voice, Chat, Picture Messaging, Camera, Bluetooth]
, Product (6): [Keyboard, Voice, Chat, Picture Messaging, Camera, WAP 2.0]
, Product (6): [Keyboard, Voice, Chat, Picture Messaging, Camera, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Video]
, Product (7): [Keyboard, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Chat, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Chat, MMS, Camera, Video, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Chat, Picture Messaging, Camera, Video, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Chat, Picture Messaging, Camera, Video, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Chat, Picture Messaging, MMS, Camera, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Chat, Picture Messaging, MMS, Camera, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Chat, Picture Messaging, MMS, Camera, Video, Bluetooth]
, Product (7): [Keyboard, Chat, Picture Messaging, MMS, Camera, Video, WAP 2.0]
, Product (7): [Keyboard, Chat, Picture Messaging, MMS, Camera, Video, WAP 1.0]
, Product (7): [Keyboard, Voice, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Voice, MMS, Camera, Video, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Voice, Picture Messaging, Camera, Video, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Voice, Picture Messaging, Camera, Video, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Voice, Picture Messaging, MMS, Camera, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Voice, Picture Messaging, MMS, Camera, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Voice, Picture Messaging, MMS, Camera, Video, Bluetooth]
, Product (7): [Keyboard, Voice, Picture Messaging, MMS, Camera, Video, WAP 2.0]
, Product (7): [Keyboard, Voice, Picture Messaging, MMS, Camera, Video, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, Camera, Video, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, Camera, Video, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Voice, Chat, MMS, Video, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, MMS, Video, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Voice, Chat, MMS, Camera, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Voice, Chat, MMS, Camera, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, MMS, Camera, Video, Bluetooth]
, Product (7): [Keyboard, Voice, Chat, MMS, Camera, Video, WAP 2.0]
, Product (7): [Keyboard, Voice, Chat, MMS, Camera, Video, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, Camera, Bluetooth, WAP 2.0]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, Camera, Bluetooth, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, Camera, Video, Bluetooth]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, Camera, Video, WAP 2.0]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, Camera, Video, WAP 1.0]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Bluetooth]
, Product (7): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, WAP 2.0]

, Product (7): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, WAP 1.0]
, Product (8): [Keyboard, Chat, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (8): [Keyboard, Chat, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 1.0]
, Product (8): [Keyboard, Voice, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (8): [Keyboard, Voice, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 1.0]
, Product (8): [Keyboard, Voice, Chat, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (8): [Keyboard, Voice, Chat, MMS, Camera, Video, Bluetooth, WAP 1.0]
, Product (8): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Video, WAP 2.0]
, Product (8): [Keyboard, Voice, Chat, Picture Messaging, Camera, Video, Bluetooth, WAP 1.0]
, Product (8): [Keyboard, Voice, Chat, Picture Messaging, Camera, Video, Bluetooth, WAP 2.0]
, Product (8): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Bluetooth, WAP 2.0]
, Product (8): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Bluetooth, WAP 1.0]
, Product (8): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Video, Bluetooth]
, Product (8): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Video, WAP 1.0]
, Product (9): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 2.0]
, Product (9): [Keyboard, Voice, Chat, Picture Messaging, MMS, Camera, Video, Bluetooth, WAP 1.0]
]

Mobile Phone CNF

Most variables contained in this CNF map to the nodes contained in the abstract syntax of the Mobile Phone FD (Figure 11.2). The correspondence with the concrete syntax is established in Figure 11.3 where the features and optional circles are decorated with the appropriate variable. Hence, we have 27 variables (1...27) and 2 auxiliary variables (28-29) generated by the encoding of the "WAP" feature (with a Boolean cardinality (1,1) and two sons) into a CNF.

```
p cnf 29 45
\\ Mobile Phone 1
-1 2 0
-1 5 0
-1 7 0
-1 3 0
 1 0
\\ Dial 2
-2 9 0
-2 10 0
-2 1  0
\\ OptMessaging 3
-3 1  0
\\ Messaging 4
```
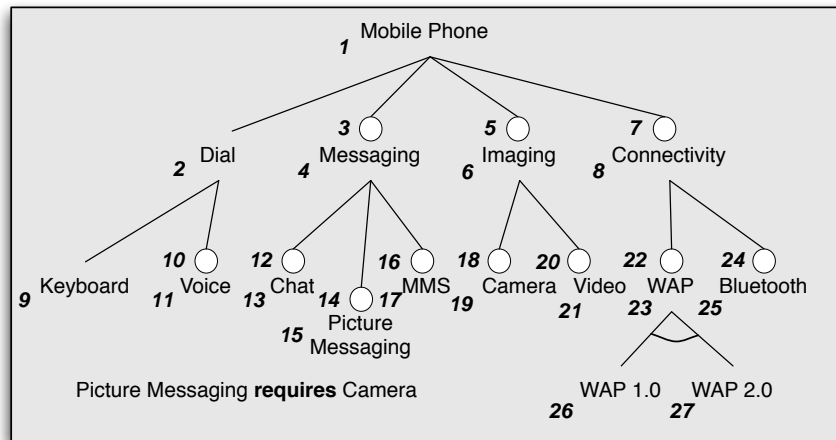
Figure 11.3: FODA FD: Mobile Phone PL

```
-4 12 0
-4 14 0
-4 16 0
-4 3  0
\\ OptImaging 5
-5 1  0
\\ Imaging 6
-6 20 0
-6 18 0
-6 5  0
\\ OptConectivity 7
-7 1  0
\\ Connectivity 8
-8 22 0
-8 24 0
-8 7  0
\\ Keyboard 9
-9 2  0
\\ OptVoice 10
-10 2  0
\\ Voice 11
-11 10  0
\\ OptChat 12
-12 4  0
\\ Chat 13
```

```
-13 12  0
\\ OptPicture Messaging 14
-14 4  0
\\ Picture Messaging 15
-15 14  0
\\ OptMMS 16
-16 4  0
\\ MMS 17
-17 16  0
\\ OptCamera 18
-18 6  0
\\ Camera 19
-19 18  0
\\ OptVideo 20
-20 6  0
\\ Video 21
-21 20  0
\\ OptWAP 22
-22 8  0
\\ WAP 23
-27 28 -23 0
-26 -28 -23 0
27 29 -23 0
26 -29 -23 0
-23 22  0
\\ OptBluetooth 24
-24 8  0
\\ Bluetooth 25
-25 24  0
\\ WAP 1.0 26
-26 23  0
\\ WAP 2.0 27
-27 23  0
\\ Picture Messaging requires Camera
-15 19 0
```

Mobile Phone DOT

```
graph FD {
 node[shape=box,style=filled];
"Mobile Phone\n(4,4)" ;
"Dial\n(2,2)" [shape=rectangle];
"OptMessaging\n(0,1)" [shape=rectangle];
```

```
"Messaging\n(3,3)" [shape=rectangle];
"OptImaging\n(0,1)" [shape=rectangle];
"Imaging\n(2,2)" [shape=rectangle];
"OptConnectivity\n(0,1)" [shape=rectangle];
"Connectivity\n(2,2)" [shape=rectangle];
"Keyboard\n(0,0)" ;
"OptVoice\n(0,1)" [shape=rectangle];
"Voice\n(0,0)" ;
"OptChat\n(0,1)" [shape=rectangle];
"Chat\n(0,0)" ;
"OptPictMessaging\n(0,1)" [shape=rectangle];
"Picture Messaging\n(0,0)" ;
"OptMMS\n(0,1)" [shape=rectangle];
"MMS\n(0,0)" ;
"OptCamera\n(0,1)" [shape=rectangle];
"Camera\n(0,0)" ;
"OptVideo\n(0,1)" [shape=rectangle];
"Video\n(0,0)" ;
"OptWAP\n(0,1)" [shape=rectangle];
"WAP\n(1,1)" [shape=rectangle];
"OptBtooth\n(0,1)" [shape=rectangle];
"Bluetooth\n(0,0)" ;
"WAP 1.0\n(0,0)" ;
"WAP 2.0\n(0,0)" ;
"Mobile Phone\n(4,4)" -- "Dial\n(2,2)";
"Mobile Phone\n(4,4)" -- "OptMessaging\n(0,1)";
"Mobile Phone\n(4,4)" -- "OptImaging\n(0,1)";
"Mobile Phone\n(4,4)" -- "OptConnectivity\n(0,1)";
"Dial\n(2,2)" -- "Keyboard\n(0,0)";
"Dial\n(2,2)" -- "OptVoice\n(0,1)";
"OptMessaging\n(0,1)" -- "Messaging\n(3,3)";
"Messaging\n(3,3)" -- "OptChat\n(0,1)";
"Messaging\n(3,3)" -- "OptPictMessaging\n(0,1)";
"Messaging\n(3,3)" -- "OptMMS\n(0,1)";
"OptImaging\n(0,1)" -- "Imaging\n(2,2)";
"Imaging\n(2,2)" -- "OptCamera\n(0,1)";
"Imaging\n(2,2)" -- "OptVideo\n(0,1)";
"OptConnectivity\n(0,1)" -- "Connectivity\n(2,2)";
"Connectivity\n(2,2)" -- "OptWAP\n(0,1)";
"Connectivity\n(2,2)" -- "OptBtooth\n(0,1)";
"OptVoice\n(0,1)" -- "Voice\n(0,0)";
"OptChat\n(0,1)" -- "Chat\n(0,0)";
"OptPictMessaging\n(0,1)" -- "Picture Messaging\n(0,0)";
```

```
"OptMMS\n(0,1)" -- "MMS\n(0,0)";
"OptCamera\n(0,1)" -- "Camera\n(0,0)";
"OptVideo\n(0,1)" -- "Video\n(0,0)";
"OptWAP\n(0,1)" -- "WAP\n(1,1)";
"WAP\n(1,1)" -- "WAP 1.0\n(0,0)";
"WAP\n(1,1)" -- "WAP 2.0\n(0,0)";
"OptBtooth\n(0,1)" -- "Bluetooth\n(0,0)";
"Picture Messaging\n(0,0)" -- "Camera\n(0,0)" [arrowhead=normal, color=blue];
"Dial\n(2,2)" -- "Chat\n(0,0)" [arrowhead=empty, color=red];
label = "Feature Diagram Mobile Phone copy"; fontsize=20}
```

Mobile Phone XML

```
<vector>
  <vfd.model.FNode>
    <NodeName>Mobile Phone</NodeName>
    <Min__Cardinality>4</Min__Cardinality>
    <Max__Cardinality>4</Max__Cardinality>
    <NbClause>0</NbClause>
    <NbTempVar>0</NbTempVar>
    <isfeature>true</isfeature>
    <dotStyle> </dotStyle>
  </vfd.model.FNode>
  <vfd.model.Edge>
    <origin reference="../../vfd.model.FNode"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>Dial</NodeName>
      <Min__Cardinality>2</Min__Cardinality>
      <Max__Cardinality>2</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>Dial</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin reference="../../vfd.model.FNode"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>OptMessaging</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
```

```
      <Max__Cardinality>1</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>false</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>OptMessaging</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin reference="../../vfd.model.FNode"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>OptImaging</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>1</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>false</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>OptImaging</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin reference="../../vfd.model.FNode"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>OptConnectivity</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>1</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>false</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>OptConnectivity</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge/destination"/>
    <destination>
      <NodeName>Keyboard</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>0</Max__Cardinality>
      <NbClause>0</NbClause>
```

```
        <NbTempVar>0</NbTempVar>
        <isfeature>true</isfeature>
        <dotStyle> </dotStyle>
      </destination>
      <answer>Keyboard</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge/destination"/>
      <destination class="vfd.model.VPNode">
        <Question></Question>
        <NodeName>OptVoice</NodeName>
        <Min__Cardinality>0</Min__Cardinality>
        <Max__Cardinality>1</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>false</isfeature>
        <dotStyle> [shape=rectangle]</dotStyle>
      </destination>
      <answer>OptVoice</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[6]/destination"/>
      <destination>
        <NodeName>Voice</NodeName>
        <Min__Cardinality>0</Min__Cardinality>
        <Max__Cardinality>0</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>true</isfeature>
        <dotStyle> </dotStyle>
      </destination>
      <answer>Voice</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[2]/destination"/>
      <destination class="vfd.model.VPNode">
        <Question></Question>
        <NodeName>Messaging</NodeName>
        <Min__Cardinality>3</Min__Cardinality>
        <Max__Cardinality>3</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>true</isfeature>
```

```
          <dotStyle> [shape=rectangle]</dotStyle>
      </destination>
      <answer>Messaging</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[8]/destination"/>
      <destination class="vfd.model.VPNode">
          <Question></Question>
          <NodeName>OptChat</NodeName>
          <Min__Cardinality>0</Min__Cardinality>
          <Max__Cardinality>1</Max__Cardinality>
          <NbClause>0</NbClause>
          <NbTempVar>0</NbTempVar>
          <isfeature>false</isfeature>
          <dotStyle> [shape=rectangle]</dotStyle>
      </destination>
      <answer>OptChat</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[8]/destination"/>
      <destination class="vfd.model.VPNode">
          <Question></Question>
          <NodeName>OptPictMessaging</NodeName>
          <Min__Cardinality>0</Min__Cardinality>
          <Max__Cardinality>1</Max__Cardinality>
          <NbClause>0</NbClause>
          <NbTempVar>0</NbTempVar>
          <isfeature>false</isfeature>
          <dotStyle> [shape=rectangle]</dotStyle>
      </destination>
      <answer>OptPictMessaging</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[8]/destination"/>
      <destination class="vfd.model.VPNode">
          <Question></Question>
          <NodeName>OptMMS</NodeName>
          <Min__Cardinality>0</Min__Cardinality>
          <Max__Cardinality>1</Max__Cardinality>
          <NbClause>0</NbClause>
          <NbTempVar>0</NbTempVar>
          <isfeature>false</isfeature>
          <dotStyle> [shape=rectangle]</dotStyle>
```

```
    </destination>
    <answer>OptMMS</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[9]/destination"/>
    <destination>
      <NodeName>Chat</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>0</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> </dotStyle>
    </destination>
    <answer>Chat</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[10]/destination"/>
    <destination>
      <NodeName>Picture Messaging</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>0</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> </dotStyle>
    </destination>
    <answer>Picture Messaging</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[11]/destination"/>
    <destination>
      <NodeName>MMS</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>0</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> </dotStyle>
    </destination>
    <answer>MMS</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
```

```
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[3]/destination"/>
      <destination class="vfd.model.VPNode">
        <Question></Question>
        <NodeName>Imaging</NodeName>
        <Min__Cardinality>2</Min__Cardinality>
        <Max__Cardinality>2</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>true</isfeature>
        <dotStyle> [shape=rectangle]</dotStyle>
      </destination>
      <answer>Imaging</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[15]/destination"/>
      <destination class="vfd.model.VPNode">
        <Question></Question>
        <NodeName>OptCamera</NodeName>
        <Min__Cardinality>0</Min__Cardinality>
        <Max__Cardinality>1</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>false</isfeature>
        <dotStyle> [shape=rectangle]</dotStyle>
      </destination>
      <answer>OptCamera</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[15]/destination"/>
      <destination class="vfd.model.VPNode">
        <Question></Question>
        <NodeName>OptVideo</NodeName>
        <Min__Cardinality>0</Min__Cardinality>
        <Max__Cardinality>1</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>false</isfeature>
        <dotStyle> [shape=rectangle]</dotStyle>
      </destination>
      <answer>OptVideo</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[16]/destination"/>
```

```
    <destination>
      <NodeName>Camera</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>0</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> </dotStyle>
    </destination>
    <answer>Camera</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[17]/destination"/>
    <destination>
      <NodeName>Video</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>0</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> </dotStyle>
    </destination>
    <answer>Video</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[4]/destination"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>Connectivity</NodeName>
      <Min__Cardinality>2</Min__Cardinality>
      <Max__Cardinality>2</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>Connectivity</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[20]/destination"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>OptWAP</NodeName>
```

```
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>1</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>false</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>OptWAP</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[20]/destination"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>OptBtooth</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>1</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>false</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>OptBtooth</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[21]/destination"/>
    <destination class="vfd.model.VPNode">
      <Question></Question>
      <NodeName>WAP</NodeName>
      <Min__Cardinality>1</Min__Cardinality>
      <Max__Cardinality>1</Max__Cardinality>
      <NbClause>0</NbClause>
      <NbTempVar>0</NbTempVar>
      <isfeature>true</isfeature>
      <dotStyle> [shape=rectangle]</dotStyle>
    </destination>
    <answer>WAP</answer>
  </vfd.model.Edge>
  <vfd.model.Edge>
    <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[22]/destination"/>
    <destination>
      <NodeName>Bluetooth</NodeName>
      <Min__Cardinality>0</Min__Cardinality>
      <Max__Cardinality>0</Max__Cardinality>
```

```
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>true</isfeature>
        <dotStyle> </dotStyle>
      </destination>
      <answer>Bluetooth</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[23]/destination"/>
      <destination>
        <NodeName>WAP 1.0</NodeName>
        <Min__Cardinality>0</Min__Cardinality>
        <Max__Cardinality>0</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>true</isfeature>
        <dotStyle> </dotStyle>
      </destination>
      <answer>WAP 1.0</answer>
    </vfd.model.Edge>
    <vfd.model.Edge>
      <origin class="vfd.model.VPNode" reference="../../vfd.model.Edge[23]/destination"/>
      <destination>
        <NodeName>WAP 2.0</NodeName>
        <Min__Cardinality>0</Min__Cardinality>
        <Max__Cardinality>0</Max__Cardinality>
        <NbClause>0</NbClause>
        <NbTempVar>0</NbTempVar>
        <isfeature>true</isfeature>
        <dotStyle> </dotStyle>
      </destination>
      <answer>WAP 2.0</answer>
    </vfd.model.Edge>
    <vfd.model.Implies>
      <origin reference="../../vfd.model.Edge[13]/destination"/>
      <destination reference="../../vfd.model.Edge[18]/destination"/>
      <answer></answer>
    </vfd.model.Implies>
</vector>
```