A Feature Modelling Language Based on Product Family Algebra

A FEATURE MODELLING LANGUAGE BASED ON PRODUCT FAMILY ALGEBRA

BY

MOHAMMED ALABBAD, B.Sc.

A THESIS
SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

© Copyright by Mohammed Alabbad, June 2013

All Rights Reserved

Master of Applied Science (2013) McMaster University
(Software Engineering) Hamilton, Ontario, Canada

TITLE: A Feature Modelling Language Based on Product Family Algebra

AUTHOR: Mohammed Alabbad
B.Sc., (Computer Science)
King Saud University, Riyadh, Saudi Arabia

SUPERVISOR: Dr. Ridha Khedri

NUMBER OF PAGES: xiii, 135

To my parents, wife, siblings, and kids to come

Abstract

Feature modelling is an emerging software engineering paradigm, which helps organizations to develop products from core assets. Products are organized into families that share common core features. Feature modelling involves capturing, into a feature model, the commonality and variability of product families and several relationships among features or products.

This thesis is about proposing a language for specifying feature models that is based on product family algebra (PFA). The language is intended to encompass the constructs found in early feature modelling graphical notations and languages. The thesis gives the syntax and the semantics of the proposed language. It discusses the design of its compiler that takes a feature model specification and generates its corresponding PFA, which can be analyzed using the tool Jory. The thesis uses a quite extensive case study to illustrate the use of the proposed language and its compiler.

Acknowledgements

I would like to thank my supervisor, Dr. Ridha Khedri, for his support and close mentoring of my work. Also, I would like to thank Qinglei Zhang and Jason Jaskolka for their valuable comments and contibutuions.

Contents

Abstract **iv**

Acknowledgements **v**

Contents **ix**

List of Tables **x**

List of Figures **xi**

1 Introduction **1**

1.1 Motivation 7

1.1.1 Shortcomings in the Current Language Used by Jory 7

1.1.2 Strengths and Weaknesses of Jory 8

1.1.3 Advantages of a High Level Language 10

1.2 Objectives 12

1.3 Thesis Structure 14

2 Background **15**

2.1 Feature Modelling 15

2.1.1 Graphical Feature Modelling 16
2.1.2 Non-Graphical Feature Modelling 22
2.2 Product Family Algebra 24
2.2.1 Set Model 26
2.2.2 Bag Model 27
2.2.3 Products, Features, Refinement, and Constraints 28
2.3 Jory Tool 30

3 Language Design **34**
3.1 Syntax and Semantics 34
3.1.1 Syntax 35
3.1.2 Semantics 39
3.2 Design Objectives 40
3.3 Design and Structure 41
3.3.1 Syntax 42
3.3.2 Semantics 47
3.4 Assessment 49
3.5 Conclusion 52

4 Design of the Compiler as a Part of the Tool Jory **53**
4.1 Compiler Structure 54
4.1.1 Flex and Bison 57
4.2 System Design 60
4.2.1 Overview 60
4.2.2 Detailed Design 62

4.3 Conclusion 69

5 Case Study **71**

5.1 E-shop System 71

5.1.1 Store Front 72

5.1.2 Business Management 76

5.2 E-shop Specification 80

5.3 Assessment and Results 83

5.3.1 Customer Service 84

5.3.2 Registration 86

5.3.3 Targeting 87

5.4 Conclusion 89

6 Conclusion **90**

6.1 Contributions 91

6.2 Future Work 92

A Feature Modelling Techniques Constructs **93**

B Sample Code: Family Generator **95**

C E-shop Specifications **100**

C.1 home_page.spec 100

C.2 registration.spec 101

C.3 registration.spec 102

C.4 wish_list.spec 105

C.5 buy_paths.spec 105

C.6 customer_service.spec . . . 109
C.7 user_behaviour_tracking.spec . . . 110
C.8 order_management.spec . . . 110
C.9 targeting.spec . . . 112
C.10 affiliates.spec . . . 114
C.11 inventory_tracking.spec . . . 114
C.12 procurement.spec . . . 115
C.13 reporting_and_analysis.spec . . . 115
C.14 external_systems_integration.spec . . . 115
C.15 administration.spec . . . 116

D Customer Service Result **117**

Bibliography **130**

List of Tables

3.1 Lexemes and Tokens of a C statement 37

3.2 Proposed Language Semantics 49

3.3 Study Summary 52

A.1 Constructs of Feature Modelling Techniques 94

List of Figures

1.1 Mobile Phone Feature Model in FODA 3

1.2 Mobile Phone Feature Model in FDL 4

1.3 Mobile Phone Feature Model in PFA 5

1.4 Equivalent Mobile Phone Feature Model in PFA 6

1.5 The Grammar of the Feature Modelling Language Used by Jory. . . . 8

1.6 Mobile Phone Feature Model in Jory's Language. 8

2.1 Mobile Phone Feature Model in FeatuRSEB 18

2.2 Mobile Phone Feature Model in Riebisch et al. 20

2.3 Mobile Phone Feature Model in CBFM 21

2.4 Mobile Phone Feature Model in TVL 24

2.5 The BDD of $f(x, y, z) = x \vee y \wedge \neg z$ 31

2.6 The Architecture of Jory 32

3.1 Calculator Grammar 39

3.2 Language Grammar 43

3.3 FODA Extension Feature Modelling Techniques 50

4.1 A General View of a Compiler 54

4.2 The Main Components of a Compiler 55

4.3 The Compiler Phases 56

4.4 Layers of Jory Tool 61
4.5 User Interface 63
4.6 A Model of the *Syntax Analysis Layer* 65
4.7 Module Uses Diagram of the Code Generator 66
4.8 Tool Illustration 69
5.1 E-shop System Overview 72
5.2 Store Front Overview 73
5.3 Home Page Overview 73
5.4 Registration Overview 74
5.5 Catalog Overview 74
5.6 Wish List Overview 75
5.7 Buy Paths Overview 75
5.8 Customer Service Overview 76
5.9 User Behaviour Tracking Overview 76
5.10 Business Management Overview 77
5.11 Order Management Overview 77
5.12 Targeting Overview 77
5.13 Affiliates Overview 78
5.14 Inventory Tracking Overview 78
5.15 Procurement Overview 79
5.16 Reporting and Analysis Overview 79
5.17 External Systems Integration Overview 80
5.18 Administration Overview 80
5.19 Customer Service on Jory After Compilation 84

5.20 Customer Service on Jory After Executing Queries Part 1 85

5.21 Customer Service on Jory After Executing Queries Part 2 86

5.22 Registration on Jory After Executing 87

5.23 Targeting on Jory After Executing 88

Chapter 1

Introduction

The software product family approach offers a solution to the demand for faster software development, lower development and maintenance costs, and higher quality software [Som01]. Product families are sets of related products in a domain that share common core assets. The process of building these core assets is called domain engineering. One of the most important steps of domain engineering is to analyze the commonalities and variabilities of product families. Studies have shown that it is advantageous to study product families in terms of features [HKM11]. Feature modelling is the process of capturing commonalities and variabilities of a product family by producing a feature model. Feature models represent the features of a system and the relationships between them. They provide a way to select features and create valid products of a family. Although feature models are related to requirements and design, all of the software development stages can benefit from feature models [KCH+90].

A system's feature is a characteristic that is visible to its stakeholders [HKM11]. Every hardware part or software artefact such as a requirement, code, or a component

can be considered as a feature [HKM06, HKM11]. For example, features in a mobile phone include elements such as the hardware component *Screen* or the software component responsible for making calls. A collection of compatible mandatory features constitute a product. We formally introduce the concepts of features and products in Chapter 2.

A product family is a set of products (might be empty) that usually share common features. The concept of product families started in the hardware industry, where manufacturers produce a variety of products by sharing most of the assets, which leads to a considerable reduction of the development cost. Product families were found to be useful in software development and made their way into the software industry [Par76]. In embedded systems, the specification of a product family can be given from a hardware perspective or a software perspective [HKM06, HKM11]. An example of product family is a mobile phone product family where all products in the family share common features such as the feature *Screen*, and differ by features such as *Keypad* and *Accessories*.

A feature model can be presented in several ways: graphically, textually, mathematically, or a combination of graphically and textually. A graphical feature model is represented as a hierarchal diagram where the root represents the system or a product family label. All the other nodes represent features or subfamilies. In graphical feature models, a feature can be composite or primitive. A composite feature (i.e., subfamily) constitutes all or some of its children. A leaf node (i.e., subfeature) is a primitive feature. A feature can be mandatory or optional. A mandatory feature

must exist in a product. Optional feature that its existence is optional. Constraints are used to eliminate unwanted features and products of a feature model, and come in two main types: inclusion and exclusion. Figure 1.1 shows an example of a graphical feature model for a mobile phone product family specified in Feature-Oriented Domain Analysis (FODA) [KCH+90]. The root is the product family labelled Mobile. It is formed of the following features: *Screen*, *Accessories*, *Keypad*, and *Calling_feature*. The features *Screen* and *Calling_feature* are mandatory features in every product of the mobile phone family. However, *Keypad* and *Accessories* are optional features indicated by hollow circles. The arc in the feature *Screen* indicates the choice between its subfeatures *Touch_Screen* and *Basic_Screen*.

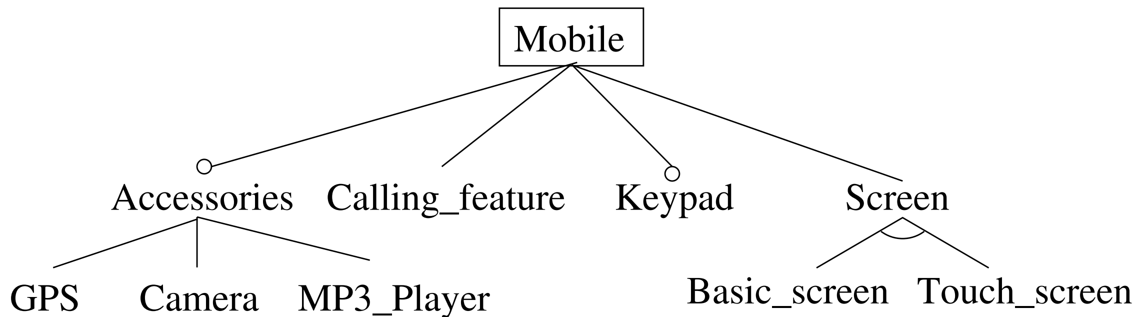

Figure 1.1: Mobile Phone Feature Model in FODA

A textual feature model consists, in most text-based feature modelling techniques, of a list of feature definitions followed by constraints. The example shown in Figure 1.2 is for the same mobile product family given in Figure 1.1 but given in Feature Description Language (FDL). It shows that the mobile phone family consists of the features *Screen*, *Accessories*, *Calling_feature*, and *Keypad*. *Screen* and *Calling_feature* are mandatory features, while *Keypad* and *Accessories* are optional features indicated

by the question mark "?". *Screen* can be only one of the two features *Basic_Screen* and *Touch_Screen*.

```
Mobile: all(Screen, Accessories?, Calling_feature, Keypad?)
Screen: one−of(Touch_Screen, Basic_Screen)
Accessories: all(GPS, Camera, MP3_Player)
```

Figure 1.2: Mobile Phone Feature Model in FDL

Feature-Oriented Domain Analysis (FODA) [KCH+90] is among the earliest proposed feature modelling techniques which is a graphical one. Most of the graphical feature modelling techniques proposed after FODA are extensions of FODA. Feature-Oriented Reuse Method (FORM) was proposed in [KKL+98] as an extension of FODA. Feature Oriented Product Line Software Engineering (FOPLE) is a refinement of FORM and FODA [KL02]. Generative Programming (GP) [CE00] and Cardinality-based Feature Models (CBFM) [CHE05] are extensions of FODA. Feature/Reuse-driven Software Engineering Business (FeatuRSEB) [GFA98] is a combination of FODA and the Reuse-driven Software Engineering Business (RESB). FeatuRSEB has three extensions, van Gurp et al. [vGBS01], Product Line Use case modelling for System and Software engineering (PLUSS) [EBB05], and Riebisch et al. [RBSP02]. Family Oriented Requirements Engineering (FORE) [Str04] extends Riebisch et al.. Feature Assembly Modeling (FAM) is a feature modelling technique that models a product family based on multiple perspectives [AZKT10].

Text-based feature modelling languages are proposed to solve the limitations of graphical feature modelling. To our knowledge, Feature Description Language (FDL) [vDK02]

was the first textual language to describe feature models. Text-based variability language (TVL) [BCFH10] is a very recent feature modelling technique which claimed to be expressive, scalable, and possessing an easy syntax.

Product family algebra (PFA) [HKM06, HKM11] is a mathematical structure of product families. It is based on the mathematical structure of an idempotent and commutative semiring. It helps capturing and analyzing the commonalities and variabilities of a product family. It also allows mathematical description and manipulation of product family specifications. We discuss PFA in detail in Section 2.2.

```
Screen = Touch_Screen + Basic_Screen
Accessories = GPS . Camera . MP3_Player
Mobile = Screen . (1 + Accessories) . (1 + Keypad) . Calling_Feature
```

Figure 1.3: Mobile Phone Feature Model in PFA

Figure 1.3 shows the mobile phone product family defined in PFA. The operator "·" indicates the mandatory presence of its operands while for instance "(1+ *Keypad*)" indicates that the feature *Keypad* is optional. *Screen* and *Accessories* are product families by themselves.

Using product family algebra, it is possible to perform calculations on features, products, and product families. Through calculus, one can verify the equivalence of two specifications, give the number of possible products of a product family, list the common features, or verify whether a family refines another family. In addition, it is possible to build new families, find new products, and exclude unwanted feature combinations [HKM06, HKM11]. To illustrate the equivalent specification of the

specification in Figure 1.3, we substitute the subfamilies *Screen* and *Accessories* by their values to get the equivalent specification shown in Figure 1.4.

Mobile = (Touch_Screen + Basic_Screen) . (1 + GPS . Camera . MP3_Player) . (1 + Keypad) . Calling_Feature

Figure 1.4: Equivalent Mobile Phone Feature Model in PFA

Based on Product Family Algebra (PFA) [HKM11] a feature modelling tool, called Jory, has been built. Jory is a feature modelling tool for product families that is based on PFA and Binary Decision Diagrams (BDDs) [Ake78]. PFA gives Jory the mathematics to specify, analyze, and calculate on product families. Moreover, PFA brings all of its benefits such as giving formal specifications for the concepts of feature, product, and family. Also, it brings the benefit of the algebraic structure and mathematical models of PFA. Jory is designed into layers to be easy to manage and improve. It implements PFA models using BDDs to handle large systems with efficiency and speed. In one megabyte of memory BDDs can handle up to $50,000$ features, and it has been tested with 2^{32} nodes [Alt10]. Since my work is an addition to Jory, I will present its design in Chapter 2.

The feature modelling tools found include AHEAD [Bat05], FaMa [BSTRc07], Feature Modelling Plugin [MC04], FeatureIDE [KTS+09], Pure::variants [Beu08], CaptainFeature [BEL03], Requiline [ML04], XFeature [CPRS04], SPLOT [MBC09], and compositional variability management framework (CVM) [APS+10]. Most of these tools do not support the conversion between feature modelling techniques. These tools support one feature modelling technique which is FODA or one of its extensions. Some tools support feature modelling as their main functionality. On the

other hand, some tools do not support feature modelling as a main functionality, but as a sub-functionality. Also, some tools add extensions and concepts that are not part of the feature modelling techniques.

1.1 Motivation

1.1.1 Shortcomings in the Current Language Used by Jory

The current language used by Jory is that of PFA which is a low level language. Figure 1.5 shows the grammar of the language. The specification written in Jory's language consists of two main parts. The first part consists of declarations of basic features. Each statement in the declaration part starts with the keyword "bf" followed by a feature name, then "%" sign followed by a comment describing the feature; this statement defines one basic feature. The first two lines of Figure 1.6 declare *Touch_Screen* and *Basic_Screen* features. The second part of the specification consists of product family definitions. The definition of a family has three parts: a family name followed by "=" sign, a product family algebra term that defines a family, "%" sign proceeding a short comment describing the family. Figure 1.6 shows an example of a product family for a mobile phone defined in Jory. The language is essentially limited to that of idempotent semiring. It does not allow concise specifications. I will elaborate further on this issue toward the end of the next subsection.

$PFA := (\langle \text{Basic_Feature} \rangle \backslash n)+\ (\langle \text{Labeled_Family} \rangle \backslash n)+$
$\langle \text{Basic_Feature} \rangle := bf \langle \text{Basic_Feature_ID} \rangle \% \langle \text{Comment} \rangle \backslash n$
$\langle \text{Labeled_Family} \rangle := \langle \text{Labeled_Family_ID} \rangle = \langle \text{Family_Term} \rangle$
$\quad \% \langle \text{Comment} \rangle \backslash n$
$\langle \text{Family_Term} \rangle := 0 | 1 | \langle \text{Basic_Feature_ID} \rangle | \langle \text{Labeled_Family_ID} \rangle |$
$\quad \langle \text{Family_Term} \rangle + \langle \text{Family_Term} \rangle |$
$\quad \langle \text{Family_Term} \rangle \dot{} \langle \text{Family_Term} \rangle$
$\langle \text{ID} \rangle := \text{Letter}(\text{Letter}|\text{Digit})+$

Figure 1.5: The Grammar of the Feature Modelling Language Used by Jory.

```
bf   touch_screen                                          %touch screen
bf   basic_screen                                          %basic screen
bf   gps                                                   %GPS feature
bf   camera                                                %Camera feature
bf   mp3_player                                            %MP3_player feature
bf   calling_feature                                       %call functionaly feature
bf   keypad                                                %keypad feature
screen = touch_screen + basic_screen                       %screen family
accessories = gps . camera . mp3_player                    %accessorise family
mobile = screen . (1+ accessories) . (1 + keypad) . call   %Mobile system
```

Figure 1.6: Mobile Phone Feature Model in Jory's Language.

1.1.2 Strengths and Weaknesses of Jory

Jory is a powerful tool that is precise and fast. It uses the theory of PFA models that are implemented using Binary Decision Diagrams (BDDs). The fast calculations using BDDs and its mathematical soundness are the source of its strength. Yet, Jory has some limitations. It lacks the support for graphical notations (i.e., the translation layer, in Jory's design, which translates between PFA and other graphical notations, is not implemented). The capability of accepting a graphical feature model as an input or generating one as an output is not enabled. Also, the translation between graphical and non-graphical feature models is not available.

The language used by Jory is a low level language: it is the language of a semiring. Writing specifications for large systems using the basic language of product family algebra is a tedious task and specifications would be too long. Moreover, many notations and terms in the language can be ambiguous for users that are unfamiliar with the PFA syntax.

When we write specifications in PFA, sometimes the existence of a feature in a product is bound to the existence of another feature. This type of dependency between features and families is specified using constraints. In other terms, constraints are used to exclude all the undesired products from the concerned family. The interpretation of constraints is not fully implemented in the Jory tool but rather passed from the interface to the BDD layer. Therefore, only one constraint can be handled at a time. The editor in the Jory Interface is not user friendly and needs improvement. Moreover, the editor does not report any syntax or semantic errors in the specification, which makes finding errors tiresome and frustrating. The tool does not include any functionality to parse the entered specification.

Jory is implemented in a way that allows it to be scalable and extendable. Therefore, solving its limitations and adding even more extensions is a feasible task. To overcome the limitations off Jory, the following solutions are suggested:

1. Designing a high level language that is close to natural languages can make the writing of specifications easier and reduce their size. Also, the high level language would enable a more comprehensive way to express constraints.

2. To make Jory more user friendly, the editor of Jory needs to be improved. Areas of improvement include hiding details that are not needed by the user, clarifying buttons and options functionalities, and separating the input and output area.

3. To support graphical notations, the implementation of the translation layer is essential. The translation layer allows the translation between different feature modelling techniques including graphical ones. Moreover, supporting graphical notations needs an editor that supports input and output in graphical format.

Suggestions 1 and 2 are in the scope of the current work. However, suggestion 3 is not under the scope of this thesis.

1.1.3 Advantages of a High Level Language

Designing a high level language has a significant number of benefits. A high level language is closer to natural languages. It would have a compiler which reports errors and handles exceptions. The compiler would also have a mechanism to recover from errors. Therefore, it is advantageous for users, specifiers, and maintainers.

A high level language is more readable and comprehensive for the user due to the following characteristics. First, the overall simplicity and naturalness of the language's syntax improves the language's readability. Second, the limited number of keywords and constructs of the language makes guessing the constructs effects an easy task and makes the user familiar with them. Moreover, the abstraction level makes the specification shorter by allowing some structures to be defined once and used multiple

times throughout a specification.

From the perspective of the specifier, writing a specification in a high level language is easier because of the simplicity of the syntax and the level of abstraction. Moreover, error reporting makes the specification more reliable and robust. Also, a high level language reduce time needed to write specifications. Expressivity means that the language has complex operators that can be expressed in a long specification in the low level language. Expressivity is a factor that shortens the specification.

One of the main costs and time consumption of a specification life cycle is maintenance and modification. Maintenance relies heavily on the readability and writability of the specification. A high level language is easier to read, thereby makeing the job of the maintainer less tedious. Error reporting and type checking also ease the maintenance and modification task.

Designing a high level language for PFA is recommended for a number of reasons. First, we can gain the benefits of a higher level language over the low level language of Jory that we discussed previously. To reach a wider community of users for PFA that might not be very comfortable with mathematical notations of the language currently used by Jory. Second, we can solve some of the limitations of Jory. In Section 2.1.2, I reported on several feature modelling languages. They use language constructs that can be adopted in the sought language.

1.2 Objectives

This thesis is about proposing a feature modelling language and designing its compiler. The sought language is based on Product Family Algebra. The compiler is expected to a part of the feature modelling tool Jory [Alt10].

The sought language combines all the benefits of a high level language, PFA, and the Jory tool. Also, it addresses the limitations of the current language that is used by Jory. To fulfil our objective, the following tasks are required:

1. Design a language by providing its syntax and semantics.
2. Build a compiler for the language.
3. Connect the compiler to the Jory tool.
4. Assess the quality of the whole system.

The high level language is required to have a simple syntax, error reporting capabilities, and an exception handling mechanism. It is intended to be easy for the user to write, read, and modify specifications. Moreover, it is intended to enable writing long and complex specifications and constraints. It is required to support the inclusion of different modules from other specification files.

The first step toward coming up with the language is to propose its design according to the above objectives. In designing the language, we need to provide the syntax and semantics of the language. The syntax of a programming language is the form of the language's expressions and statement units. The semantics is the meaning of

those expressions and statement units. The clarity and precision of the design is an important factor for the success of the language.

The smallest syntactic units of the language are tokens which are used to specify lexemes. Lexemes include identifiers, operators, and keywords. Tokens are described using regular expressions. The expressions and statement units which consist of one or more tokens are described using context-free grammar or Backus Naur Form (BNF). The grammar of the language is discussed in detail in Section 3.3.

The second step is designing and implementing a compiler for the language. The compiler accepts the PFA specification written in the high level language, checks the syntax of the specification, reports errors if there are any, and generates the output specification in Jory's language. The compiler is to be connected to Jory to generate the final result of the specification.

The compiler validates the syntax of specifications. It keeps track of the identifiers of the specifications, reports errors and their positions, then translates the specification to the equivalent specification in the target language (i.e., the language of product family algebra). The structure of the compiler comes in two parts: front-end (analysis) and back-end (synthesis). The main functionality of the front-end deals with the input specification. It validates the syntax of the input, then generates an intermediate representation of the specification. The back-end functionality deals with the target specification by taking the intermediate representation as an input and generating the equivalent target specification. The compiler is discussed in detail in

Section 4.1. Connecting the compiler to Jory brings the benefits of both the compiler and Jory in the form of one black box with hidden details from the user. In this way, the user writes the specifications and obtains the result in the same interface.

The last step is to assess the system. In order to validate the system and ensure that it meets the expectations and generates the correct results, I focus on testing. Testing involves sub-system testing and whole system testing. The testing will be carried out on different levels and aspects. The test plan involves unit testing up to system testing. Unit testing assesses the smallest code units. Sub-system testing is the test performed on the level of the major components of the system. System testing is done on the level of the whole system. This involves providing the system with an input and validating the generated output with the expected output.

1.3 Thesis Structure

In Chapter 2, I give the needed background. We consider the concepts presented in this chapter in more detail. In Chapter 3, we consider the design of the proposed language. In Chapter 4, I present the design and implementation of the language's compiler. In Chapter 5, I present a case study to assess and validate the system. In Chapter 6, I conclude and point to future work.

Chapter 2

Background

In this chapter, I present the background behind this work. In Section 2.1 we study feature modelling techniques in more detail. In Section 2.2, we examine PFA and its models. In Section 2.3, we consider the Jory tool design.

2.1 Feature Modelling

The basic elements in a feature model include features, products, product families, and constraints. In Chapter 1, we have seen their definitions with examples of a mobile phone product family. We have introduced feature modelling. We have also discussed that depending on the feature modelling technique, feature models can have many forms such as graphical, textual, and combination of graphical and textual. In this section, we discuss graphical and non-graphical feature modelling techniques in more detail.

2.1.1 Graphical Feature Modelling

Graphical feature modelling techniques represent a product family as a graph. The model is a hierarchal tree with the root node representing the name of the system. Every other node represents a feature or a subfamily. A graphical feature model can be an *and/or tree* or a *directed acyclic graph (DAG)*.

One of the drawbacks of graphical feature modelling can be perceived when we handle large systems. It becomes hard to visualize the product family because the resulting model is so large. Therefore, it becomes difficult and tedious to understand the model. Finding features and their relationships in the model becomes a difficult task for stakeholders. Also, counting the different products of the family from the model would be nearly impossible.

After analyzing and studying the literature, we can classify graphical feature modelling techniques in four classifications: basic feature models, constraints feature models, cardinality-based feature models, and extended feature models.

Basic Feature Models

This group is basically the feature modelling technique FODA and its successors which have added minor extensions to it. The main characteristics of this group of feature models are the following:

- The feature diagram is an *and/or tree* as in FODA or a *DAG* as in FORM.
- The root node represents the system label or the product family name.

- Leaf nodes denote basic or primitive features (i.e., undividable features).
- All other nodes which are not the root or leaf nodes, represent composite features or subfamilies. A composite feature has subfeatures and a relation between the parent and its subfeatures.
- Features can be mandatory or optional. A mandatory feature means that if its parent is selected in a product then it has to be selected as well. The root feature is mandatory in all products. Optional features are those such that if their parent features are selected then their existence is optional.
- The relations between a parent feature and its subfeatures are one of the following relations: AND (i.e., if the parent feature is selected then all of its subfeatures are selected), OR (i.e., one or more subfeatures can be selected), or XOR (i.e., exclusive-or, only one subfeature is selected if the parent is selected).
- Constraints are a textual presentation. Constraints are one of two forms: include (i.e., if feature A is selected then feature B must be selected), and exclude (i.e., if feature A is selected then feature B is excluded).

FORM and FOPEL techniques extend FODA with four layers. Each feature belongs to one of the following layers: capability layer, operating environment layer, domain technology layer, and implementation technique layer. They also add more types of relations which are *generalization/specialization* and *implemented-by*.

The most basic feature modelling technique is FODA. In Figure 1.1, we saw an example of a model for a mobile phone product family specified in FODA. The feature diagram is an *and/or tree*. The root node is enclosed in a box and has the product

family name. *Screen* and *Accessories* are composite features while the rest are leaf nodes. *Keypad* and *Accessories* are optional features indicted by the hollow circle above them; the other features are mandatory features. The relation between *Screen* and its subfeature is an XOR relation. The relations between the root node and *Accessories* and their subfeatures are AND relations.

Constraints Feature Models

Feature models in this group extended the basic feature models with representing the constraints graphically in the diagram. Include and exclude constraints are represented as lines connecting the features on the feature diagram. This extension to FODA was added in FeatuRSEB and Van Gurp. Other feature modelling techniques have even more complex constraints of the form: "if features A and B are selected then feature C is selected."

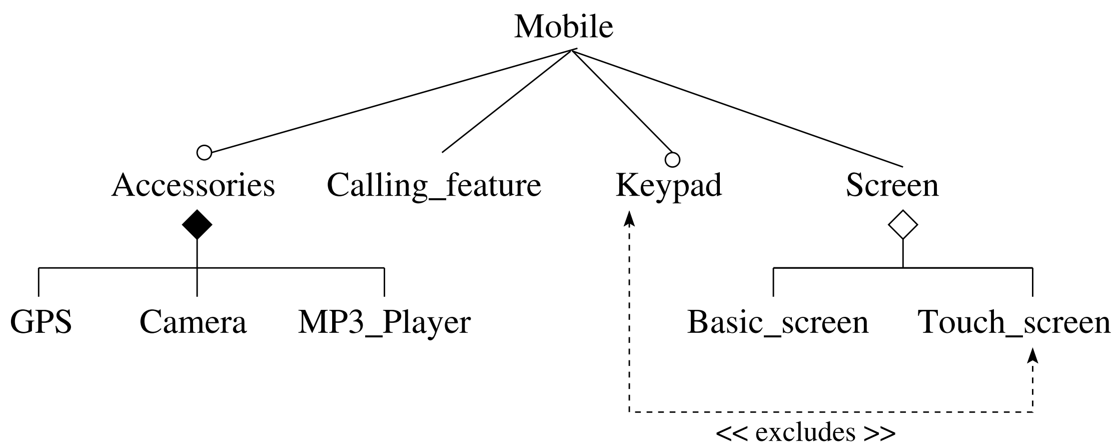

Figure 2.1: Mobile Phone Feature Model in FeatuRSEB

Figure 2.1 shows the mobile phone product family specified using the FeatuRSEB

feature-modelling technique. The main difference with FODA is the existence of the *excludes* constraint in the feature diagram. In the mobile phone product family, the existence of *Keypad* excludes the existence of *Touch_Screen* and vice versa. In Figure 2.1, the relationship between *Screen* and its subfeatures is an XOR relation indicated by the white diamond. On the other hand, the relation between *Accessories* and its children is an OR relationship represented by the black diamond.

Cardinality-Based Feature Models

Cardinality-based feature models extend constraints feature models with the following notations:

- Feature cardinality represents the number of occurrences of a feature in a product family. It is of the form $\langle n - m \rangle$, where n is the least number of occurrences of a feature in a product and m is the maximum number of occurrences of the feature. Mandatory and optional features can be presented using feature cardinality. A mandatory feature can be presented as $\langle 1 - 1 \rangle$, and the optional feature as $\langle 0 - 1 \rangle$.

- Group cardinality indicates the number of subfeatures to be selected when the parent feature is selected. Group cardinality is of the form $n\,..\,m$ where n indicates the minimum number of selected subfeatures while m represents the maximum number of selected subfeatures. The relationships AND, OR, and XOR can be considered as special cases of group cardinality. For example, the AND relation is the cardinality $n\,..\,n$ where n is the number of subfeatures. The OR relation is the cardinality $1\,..\,m$ where m is the number of subfeatures. The

XOR relation is the cardinality 1 .. 1 where only one subfeature is selected.

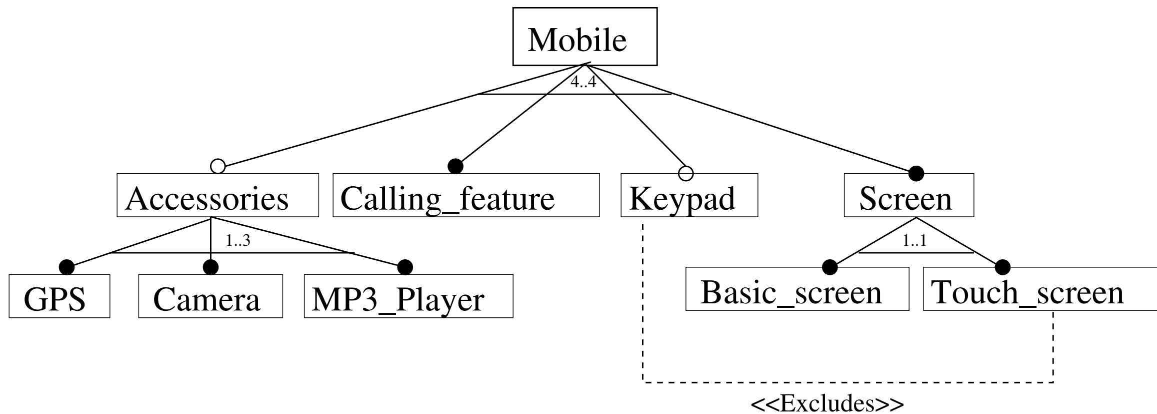

Figure 2.2: Mobile Phone Feature Model in Riebisch et al.

In Figure 2.2, we see the feature diagram in the feature modelling technique Riebisch et al. for the mobile phone product family. We notice that all features are boxed. Also, mandatory features are denoted by black circles and optional features denoted by white circles. Moreover, we changed all the relationships between composite features and their subfeatures to be alternative relations with cardinalities. The first one is the relation between the root node and its children. The cardinality of this relation is 4 .. 4 where the minimum and maximum bounds are the same and equal to the number of children. This means all features have to be selected in all products and this is an alternative of the relation AND. The second relation is between *Screen* and its subfeatures. The cardinality is 1 .. 1 which means only one feature can be selected which is equivalent to XOR relation. The third relation between *Accessories* and its subfeatures, which has the cardinality 0 .. 3, means the number of selected features is between 0 and 3.

Extended Feature Models

This group of feature models extend the previous models with attributes or diagram reference or both. Attributes were proposed to present a choice of value from a large domain such as: Integer or Float. To model this, a feature is associated with a type and a value. The type comes between parentheses after the feature name (i.e., *featurename(type)*), and if the feature has a value, then it comes after the type (i.e., *featurename(type: value)*). If a feature has multiple attributes, then they are modelled as subfeatures. An attribute is represented on the model graph. Feature models use diagram references as a way of abstraction and reuse of an already created models. The feature model would have a special node which refers to a model.

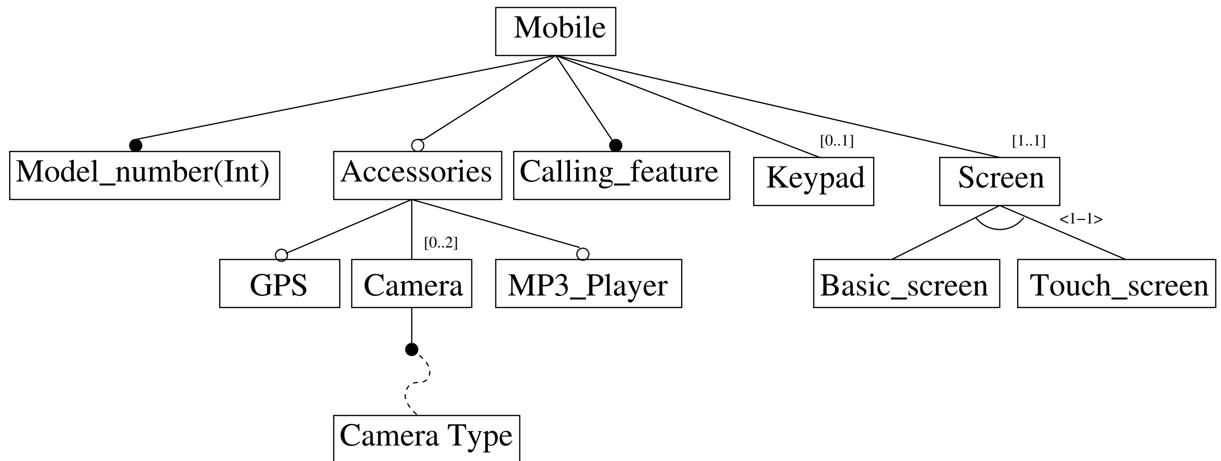

Figure 2.3: Mobile Phone Feature Model in CBFM

Figure 2.3 shows the mobile phone product family specified using the CBFM feature-modelling technique. There are three main extensions to the previous model in Figure 2.2. First, feature cardinalities are added to the features *Screen*, *Keypad*, and *Camera*. Second, a feature attribute is added to the new feature *Model_number* which

means it has an attribute of type integer. Third, the model refers to another model using diagram reference. The feature *Camera* refers to another model which contains the type of *Camera*.

2.1.2 Non-Graphical Feature Modelling

The non-graphical feature modelling techniques include textual and algebraic feature modelling techniques. Textual feature modelling techniques model product families in forms of text rather than diagrams. On the other hand, PFA is an algebraic feature modelling technique that was introduced to empower feature modelling with algebraic functionalities such as calculating ways to expand feature modelling and abstraction.

Feature modelling diagrams are not sufficient to be used in industrial settings for many reasons. This led to the creation of text-based feature modelling techniques. Textual feature modelling techniques solve some of the limitations and drawbacks of graphical feature modelling techniques. Moreover, they have a better abstraction mechanism. They are easier to read and write by humans because they are close to natural languages. A textual feature model can be written in a normal text editor without the need for specific tools. However, a textual feature model could be long and hard to follow in large scale feature models. Also, some of the textual techniques assume some programming background for the user. Moreover, manipulation of textual feature models cannot be automated.

Feature Description Language (FDL) [vDK02] is a textual language for describing

feature models. FDL claims to be easy to communicate through emails and to be translated easily to UML models and subsequently to actual code. FDL feature models consists of two parts: feature definitions and constraints. The first part consists of feature definitions, each of which starts by a feature name followed by an expression. An expression could be one of the following: atomic features, mandatory features, alternative features, or non-exclusive feature groups. The constraints part contains diagram constraints and user constraints. Diagram constraints are of the form "f1 requires f2". User Constraints are of the form "include f1". In Figure 1.2, we have seen the mobile phone product family specified using the FDL feature-modelling technique.

Another textual feature modelling technique is the Text-based Variability Language (TVL). The goal of this language is to handle large scale models. It is presented as an alternative and complementary to the graphical models. TVL is a C-like programming language. The root feature starts with the word "root", followed by the feature name then the composed features. A decomposition is denoted by "group" followed by "allOf" for mandatory features, "oneOf" for alternative features, "someOf" for non-exclusive features, or the group cardinality. An optional feature is denoted by "opt". A feature that is already a member of a family can be reused in a different family proceeded by the keyword "shared". Moreover, features in TVL can have attributes with types associated to them. Constraints in TVL are boolean expression that can be added to the body of the feature. Figure 2.4 shows the mobile phone product family specified using the TVL feature-modelling technique.

```
root Mobile group allOf {
  Screen group oneOf {
    Basic_Screen ,
    Touch_Screen
    },
  opt Keypad,
  Calling_feature ,
  opt Accessories group someOf {
    GPS,
    Camera,
    MP3_Player
    }
  }
```

Figure 2.4: Mobile Phone Feature Model in TVL

2.2 Product Family Algebra

Product family algebra (PFA) is an algebraic feature modelling technique. PFA is based on the mathematical structure of an idempotent and commutative semiring which gives it the power to describe product families precisely. Moreover, it allows algebraic calculations and manipulations of product families to generate new information about those product families.

In Figure 1.3, we have seen an example of a specification written in PFA for a mobile phone product family. In the following, we introduce the mathematical foundations of product family algebra in more detail.

Let S be a set, and let $+$ and $\cdot$ be two binary operations on S (i.e., $+, \cdot : S \times S \to S$). We say that:

The operation $\cdot$ is associative $\iff \forall(x, y, z \mid x, y, z \in S : x \cdot (y \cdot z) = (x \cdot y) \cdot z)$.

The operation $\cdot$ is commutative $\iff \forall(x, y \mid x, y \in S : x \cdot y = y \cdot x)$.

The operation $\cdot$ is idempotent $\iff \forall(x \mid x \in S : x \cdot x = x)$.

The element 1 is the identity element for $\cdot \iff \forall(x \mid x \in S : x \cdot 1 = x = 1 \cdot x)$.

The element 0 is an annihilator for $\cdot \iff \forall(x \mid x \in S : x \cdot 0 = 0 = 0 \cdot x)$.

The operation $\cdot$ right-distributes over the operation $+ \iff \forall(x, y, z \mid x, y, z \in S : (x + y) \cdot z = x \cdot z + y \cdot z)$.

The operation $\cdot$ left-distributes over the operation $+ \iff \forall(x, y, z \mid x, y, z \in S : x \cdot (y + z) = x \cdot y + x \cdot z)$.

Definition 2.2.1. A *semigroup* is an algebraic structure $(S, \cdot)$ such that S is a set, and $\cdot$ is an associative binary operation. A semigroup is commutative if $\cdot$ is commutative, and idempotent if $\cdot$ is idempotent.

Definition 2.2.2. A *monoid* is an algebraic structure $(S, \cdot, 1)$, such that $(S, \cdot)$ is a semigroup and 1 is the identity element.

Definition 2.2.3 ([HKM11])**.** A *semiring* is an algebraic structure $(S, +, 0, \cdot, 1)$, such that $(S, +, 0)$ is a commutative monoid and $(S, \cdot, 1)$ is a monoid such that $\cdot$ distributes over $+$ and 0 is an annihilator for $\cdot$. A semiring is idempotent if $+$ is idempotent and commutative if $\cdot$ is commutative. In an idempotent semiring the relation $a \leq b \Leftrightarrow_{df} a + b = b$ is a partial order (i.e., a reflexive, antisymmetric and transitive relation) called the natural order on S. It has 0 as its least element. Moreover, $+$ and $\cdot$ are isotone with respect to $\leq$.

Definition 2.2.4 ([HKM11])**.** *Product family algebra* is an idempotent and commutative semiring $(S, +, 0, \cdot, 1)$ in which 1 is a product. Its elements are product families. A family g is a subfamily of a family f iff $g \leq f$, where $\leq$ is the natural semiring order.

In product family context, addition + can be interpreted as a choice or option between two product families as an OR relation in feature diagrams, and multiplication $\cdot$ as a mandatory composition of features as an AND relation. The constant 0 represents the empty family and the constant 1 represents the family that has one product without features. The term $a + 1$ is the product family offering the choice between a and the identity product and indicates that the feature a is optional.

2.2.1 Set Model

To give a meaning for the semiring structure, concrete models have been proposed in [HKM06]. In the model that we discuss in this section, a product is a set of features and a family is a set of sets of features.

Let $\mathbb{F}$ be a set of arbitrary *features*. A *product* is a member of the power set of $\mathbb{F}$. The set of all possible products is $\mathbb{P} =_{df} \mathcal{P}(\mathbb{F})$, the set of all subsets of $\mathbb{F}$. The special element $1 = \{\varnothing\}$; the family of one product that has no features. The special family 0 is the empty set.

The definition of the semiring operations on the set model are given in [HKM11]. The operation $\cdot$ offers the composition between families.

$$\cdot : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) \to \mathcal{P}(\mathbb{P})$$

$$P \cdot Q =_{df} \{p \cup q \mid p \in P \land q \in Q\}.$$

The operation + offers the choice between families.

$$+ : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) \rightarrow \mathcal{P}(\mathbb{P})$$
$$P + Q =_{df} P \cup Q,$$

where $\cup$ denotes the set union.

The structure of the set model is $\mathbb{P}\text{FS} =_{df} \left(\mathcal{P}(\mathbb{P}), +, \varnothing, \cdot, \{\varnothing\}\right)$. As we can see from the above representation of the model, the set model does not allow multiple occurrences of a feature in a product.

2.2.2 Bag Model

Sometimes, we need to have multiple occurrences of the same feature in a product. To do this, we use the bag model. The definition of 1, 0, and the operation + are exactly the same as in the set model. However, we define the operation $\cdot$ as the composition of families.

$$\cdot : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) \rightarrow \mathcal{P}(\mathbb{P})$$
$$P \cdot Q =_{df} \{p \uplus q \mid p \in P \wedge q \in Q\},$$

where $\uplus$ denotes the bag sum.

2.2.3 Products, Features, Refinement, and Constraints

Definition 2.2.5 ([HKM11])**.** Assume a commutative idempotent semiring A, and an element $a \in A$. We say a is a product if it satisfies:

$$\forall (b \mid: b \leq a \implies (b = 0 \vee b = a)), \tag{2.1}$$

$$\forall (b, c \mid: a \leq b + c \implies (a \leq b \vee a \leq c)). \tag{2.2}$$

The element 0 is a product. A product is said to be proper if $a \neq 0$.

Equation 2.1 indicates that a product does not have a subfamily except the empty family and itself. Equation 2.2 states that if a product a is a subfamily of a family formed by c and b, it must be a subfamily of one of them. An example of a product is Mobile specified as follows:

$$\text{Mobile} = \text{Screen.Keybad.Calling_feature}$$

Mobile is a product according to Definition 2.2.5.

Definition 2.2.6 ([HKM11])**.** We say that a is a feature if it is a proper product different than 1 satisfying:

$$\forall (b \mid: b | a \implies b = 1 \vee b = a), \tag{2.3}$$

$$\forall (b, c \mid: a | (b \cdot c) \implies a \mid b \vee a \mid c), \tag{2.4}$$

where the division operator $|$ is defined by $x \mid y \iff_{df} \exists (z \mid: y = x \cdot z)$.

Equation 2.3 states that if we have a product b that divides a, then either b is 1 or $b = a$. Equation 2.4 states that for all product families b and c, if a is mandatory to form $b \cdot c$, then it is mandatory to form b or it is mandatory to form c. Some examples of features in the mobile phone product family are the features *Screen* and *Keypad*.

Definition 2.2.7 ([HKM11])**.** The *refinement* is the relation defined as

$$a \sqsubseteq b \Longleftrightarrow_{df} \quad \exists (c \mid : a \leq b \cdot c).$$

This refinement relation $a \sqsubseteq b$ means that every family in a has all features of some families in b. It is a preorder (i.e., reflexive and transitive).
We consider families *NewMobile* and *OldMobile* that are defined as follows:

NewMobile = Screen · Keypad · Calling_feature · (GPS + Camera + MP3_Player)

OldMobile = Screen · Keypad · (GPS + Camera + MP3_Player)

The family $NewMobile$ has the following three products:

Screen · Keypad · Calling_feature · GPS

Screen · Keypad · Calling_feature · Camera

Screen · Keypad · Calling_feature · MP3_Player

The family OldMobile also has three products:

Screen · Keypad · GPS

Screen · Keypad · Camera

Screen · Keypad · MP3_Player

We say $NewMobile$ refines $OldMobile$ and we write $NewMobile \sqsubseteq OldMobile$ because every product in $NewMobile$ has all features of some products in $OldMobile$.

Definition 2.2.8 ([HKM11])**.** Assume a product family algebra. For elements a, b, c, d, and product p we define *constraint*, in a family-induction style,

$$a \xrightarrow{p} b \Longleftrightarrow_{df} (p \sqsubseteq a \implies p \sqsubseteq b),$$
$$a \xrightarrow{c+d} b \Longleftrightarrow_{df} a \xrightarrow{c} b \wedge a \xrightarrow{d} b,$$

where $a \xrightarrow{e} b$ means that if product e has feature a then it also has feature b.

Constraints have two main usages: inclusion or exclusion. Consider the following features for a mobile phone: *Screen*, *Keypad*, *Touch_Screen*, *Battery*, and *Memory*. An example of an inclusion constraint is: $Screen \xrightarrow{Mobile} Keypad$, which means if we have a *Screen* in mobile phone product then we must have a *Keypad*. On the other hand, an example of an exclusion constraint comes in the form: $Screen \cdot Touch_screen \xrightarrow{Mobile} 0$, which states that we cannot have a *Screen* and a *Touch_Screen* at the same time in a mobile phone product.

2.3 Jory Tool

In Chapter 1, I introduced the tool Jory. We have seen that it is based on PFA and BDDs. In this section, we look into the detailed design of the tool.

Jory implements PFA models (i.e., set and bag) using Binary Decision Diagrams (BDDs). BDDs are used to handle large systems with efficiency and speed. BDDs are directed acyclic graphs (DAGs) representing binary functions as binary trees. The nodes represent boolean variables while the terminal nodes are 0 or 1. Every node has two children. One is connected by a low-edge (denoted by ⇢) while the other is connected by a high-edge (denoted by →). Jory uses Ordered Binary Decision Diagrams (ROBDDs) which always has the nodes ordered. Nodes are unique, which means that is no two nodes at the same level have the same label.

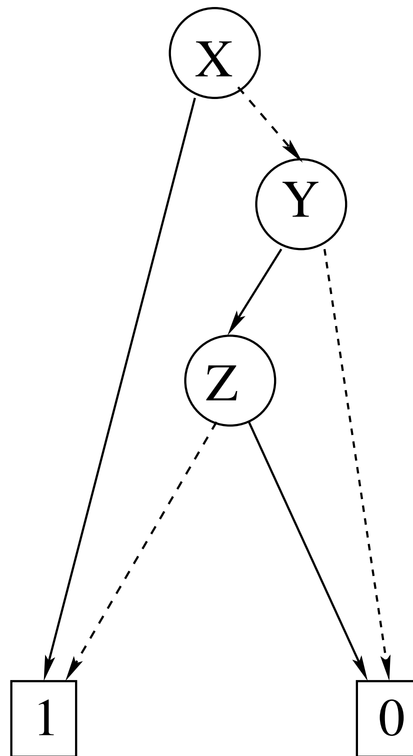

Figure 2.5: The BDD of $f(x, y, z) = x \vee y \wedge \neg z$

Figure 2.5, which is borrowed from [Alt10], shows the BDD representation of the function $f(x, y, z) = x \vee y \wedge \neg z$. The function f is a boolean function that evaluate

to true or false. The nodes x, y, and z are boolean variables. If the boolean variable evaluates to true then the high-edge is selected, otherwise, the low-edge is selected. The boolean operators $\vee$, $\wedge$, and $\neg$ denote *or*, *and*, and *negation*, respectively. When the function evaluates to *true*, then the path of nodes leads to 1, and we say the function is satisfied.

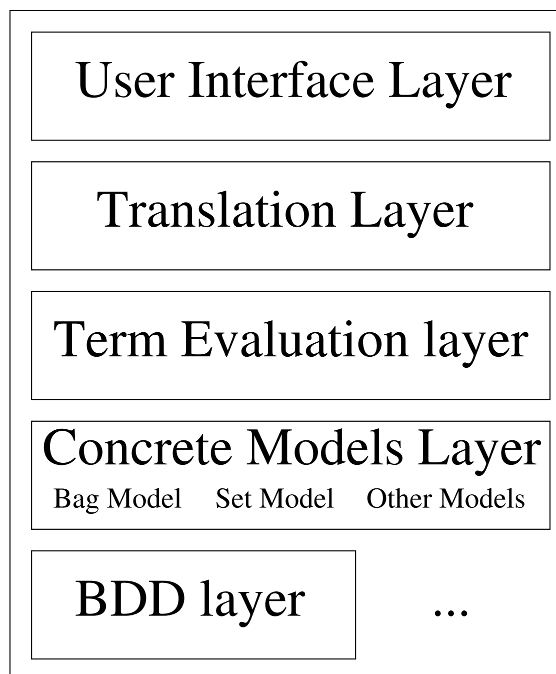

Figure 2.6: The Architecture of Jory

Jory is designed using a layered architecture. Generally, a layered architecture is best to design, implement, maintain, and modify layers separately. Jory consists of five layers: *User Interface Layer*, *Translation Layer*, *Term Evaluation Layer*, *Concrete Model Layer*, and *BDD Layer*. Figure 2.6 shows the architecture of Jory.

The *User Interface Layer* includes all the modules that make the user interface of the

system. The user interface is where the user inputs the specification written in the language of PFA. Also, it allows the user to set the configurations and the environment and select the preferences. The *User Interface Layer* is implemented in Python.

The *Translation Layer* translates from one feature modelling technique to another. It is used by Jory to translate from the different modelling techniques to PFA to be used in lower layers, and from PFA to other modelling techniques to be used in upper layers. This layer is not currently implemented and is intended for future work.

The *Term Evaluation Layer* takes PFA specification, evaluates it, analyzes it, and generates a registry of the features, products, and families. This layer is implemented in Haskell.

The *Concrete Model Layer* uses the registry generated from the *Term Evaluation Layer* to handle the set and bag models. It generates the BDD code in the selected model.

The *BDD Layer* is where the BDD code generated from the concrete model layer gets executed. Also, it is where the specification, queries, and transactions take place, and also where the result is produced.

Chapter 3

Language Design

The aim of this chapter is to discuss the main issues that are faced when designing the proposed language. In Section 3.1, we examine syntax and semantics of programming languages. In Section 3.2, we list the objectives of the proposed language design. In Section 3.3, we present the actual design and structure of the language. In Section 3.4, we assess the language with regard to the design objectives. Finally, Section 3.5 concludes the chapter.

3.1 Syntax and Semantics

Language design involves presenting the constructs of the language and their intended effects. Therefore, language design involves the discussion of two aspects of the language: its syntax and its semantics. The syntax is the format of the language statements and expressions while the semantics is the meaning or effect of these statements and expressions [Seb01].

3.1.1 Syntax

The syntax of a programming language defines the set of acceptable strings. It is usually articulated using a combination of regular expressions and context-free grammars. Regular expressions are used to describe the lexical units of tokens while context-free grammars, usually given under Backus-Naur Form (BNF), are used to describe the grammatical structure of the language.

A regular expression is a sequence of symbols that are used to represent or recognize a regular set of strings. If Σ is a finite alphabet set, a regular expression represents a (possibly infinite) set of strings in Σ^*. Regular expressions are defined by induction. The basic regular expressions are the following:

- $a, \ \forall a \in \Sigma$: It recognizes the symbol a only. Therefore, $L(a) = \{a\}$;
- ϵ: It recognizes only ϵ (the null string). Therefore, $L(\epsilon) = \{\epsilon\}$;
- $\varnothing$: It recognizes nothing. Therefore, $L(\varnothing) = \{\ \}$;

Compound regular expressions are defined inductively from basic regular expressions and binary operators $+$ and $\cdot$ and unary operators *, $^+$, and $?$. If σ and β are two regular expressions, then:

- $\sigma + \beta$ or $\sigma \mid \beta$ are regular expressions that express the choice between σ and β. For example, $ab + ba$ recognizes the string ab or the string ba.
- $\sigma \cdot \beta$ or $\sigma\beta$ are regular expressions that recognize the set of strings in the language obtained by the concatenation of $L(\sigma)$ with $L(\beta)$. For example, ab recognizes the string given by a followed by b.

- σ^* is a regular expression that recognizes the set of strings in the language obtained by the concatenation of $L(\sigma)$ repeated zero or more times. For instance, a^* recognizes the set of strings $\{\epsilon, a, aa, ...\}$.

- σ^+ is a regular expression that recognizes the set of strings in the language obtained by the concatenation of $L(\sigma)$ repeated one or more times. For instance, a^+ recognizes the set of set of strings $\{a, aa, aaa, ...\}$.

- $\sigma?$ is a regular expression that recognizes the set of strings in the language obtained by the concatenation of $L(\sigma)$ repeated zero or one time. For instance, $a?$ recognizes the strings ϵ and a only.

In regular expressions, unary operators have higher precedence than binary operators. The operator $\cdot$ has a higher precedence than +. Moreover, parentheses can be used to delimit the scope of an operator. For example, the regular expression $(a \mid b)\,c$ recognizes the strings ac or bc. Also, square brackets are used to recognize one character between the brackets. For example, $[abc]$ recognizes a, b, or c. To recognize a rang of characters or numbers, we use $-$. For instance, $[a - z]$ recognizes a lower case character in the range a to z.

Regular expressions are used to describe tokens and lexemes in programming languages. For example, the expression used to describe identifiers is $[a - z\,A - Z][_a - z\,A - Z\,0 - 9]^*$ which starts with a lower case or an upper case letter followed by "_", a lower case letter, an upper case letter, or a digit, repeated zero or more times.

Lexical Structure

In programming language syntax, a lexical structure is described as a set of regular expressions. These regular expressions define the set of acceptable sequences of characters also known as tokens. Tokens are used to categorize lexemes which include identifiers, operators, and reserved words. Lexemes are the actual value of tokens. Tokens can refer to multiple lexemes. For example, an *identifier* token can be any string that forms an identifier like "name" or "id". However, a token can refer to a specific lexeme. For instance, a *plus* token can only refer to "+" in a specific language. The following is an example of tokens and lexemes of a C statement: if $(a == 5)$ $x = 6$;

Lexem	**Token**	**Lexem**	**Token**
if	IF	)	CP
(	OP	x	ID
a	ID	=	Equal
==	is_Equal	6	DIGIT
5	DIGIT	;	SC

Table 3.1: Lexemes and Tokens of a C statement

Table 3.1 shows the lexemes and tokens of the previous C statement. "if" is a keyword and its token is "*IF*". The tokens for open and close parentheses are "*OP*" and "*CP*", respectively. The identifier "a" and "x" have the token "*ID*" and the lexeme of each is the actual name "a" and "x". Also, "5" and "6" share the token "*DIGIT*" and differ in the lexeme or value. The assignment operator "=" has the token *Equal* while the token *is_Equal* is for the comparison operator "==".

Grammatical Structure

To describe the syntax of a programming language, we need to write the grammar of that language. Grammar specifies the structure of the program and the acceptable form of statements and expressions. BNF is a notation used to describe and define programming language grammars. A grammar written in BNF is a finite set of rules or productions defining the expressions and statements of the language.

A BNF rule is of the form $\langle \text{LHS} \rangle \longrightarrow \langle \text{RHS} \rangle$, where LHS is a non-terminal or an abstraction name. RHS is the definition of the abstraction which contains terminal "tokens" and non-terminals. A non-terminal, which is given between two angle brackets (i.e., $\langle\ \rangle$), generates strings and tokens and is defined in another rule. A terminal is the actual string or token and has no further definitions. For example, the BNF rule of an expression can be as follows:

$$\langle \text{EXPR} \rangle \longrightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle$$

The expression is defined as an expression followed by a plus token, then another expression. A non-terminal or abstraction can have multiple definitions. They can be grouped into one definition separated by a "|" symbol.

A BNF rule can be used to describe lists. For this purpose, the rule needs to be recursive by having its name in the RHS. For example, the following rule is used to describe a list of identifiers.

$$\langle \text{ID_LIST} \rangle \longrightarrow \text{ID} \mid \text{ID}, \langle \text{ID_LIST} \rangle$$

A list of identifiers is defined as a single identifier, or an identifier followed by a comma, followed by a list of identifiers. Using this rule, a list of one or more identifiers can be articulated.

$$
\begin{aligned}
&\langle \text{PROGRAM} \rangle \longrightarrow \langle \text{EXPR} \rangle \\
&\langle \text{EXPR} \rangle \longrightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\
&\langle \text{TERM} \rangle \longrightarrow \langle \text{TERM} \rangle * \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\
&\langle \text{FACTOR} \rangle \longrightarrow (\langle \text{EXPR} \rangle) \mid \langle \text{NUMBER} \rangle \\
&\langle \text{NUMBER} \rangle \longrightarrow \langle \text{DIGIT} \rangle \mid \langle \text{DIGIT} \rangle \langle \text{NUMBER} \rangle \\
&\langle \text{DIGIT} \rangle \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 3.1: Calculator Grammar

In Figure 3.1, we consider a simple BNF grammar for a calculator to illustrate the usage of BNF to articulate a grammar. In this illustrative grammar, $\langle \text{PROGRAM} \rangle$ is the start symbol. The first rule of the grammar gives the structure of the program, which is an expression $\langle \text{EXPR} \rangle$. The second rule specifies an expression which is either $\langle \text{EXPR} \rangle + \langle \text{TERM} \rangle$ or $\langle \text{TERM} \rangle$. A term is $\langle \text{TERM} \rangle * \langle \text{FACTOR} \rangle$ or $\langle \text{FACTOR} \rangle$. A factor is an expression enclosed in parentheses or a number. A number a digit repeated one or more times. A digit is within $[0 - 9]$.

3.1.2 Semantics

The semantics of a programming language is the meaning of its constructs. There is no universally accepted method for describing semantics. Some methods have been developed such as operational semantics, axiomatic semantics, or denotational semantics [Seb01]. In practice, natural language descriptions are used to articulate the meaning and expected behaviour of the language constructs.

3.2 Design Objectives

In this section, we consider the design objectives of the proposed language. The sought language has to be *readable, writable, modular, expressive, and have a deterministic interpretation* on the domain of use. These characteristics of the proposed language overlap and contribute to one another. For example, the simplicity of a language improves its readability and writability. In the following, we consider the objectives one by one.

Readability

The objective of readability means that the language is easy to be read and understood by general users. The readability of a language plays a big role in the whole development life cycle, especially the maintenance phase. Therefore, it enhances the quality of a specification or a program written in the language.

Writability

We aim for our proposed language to be writable. This means that it should be easy to write a specification in the language. A language is writable when it is simple. Language simplicity can be achieved by limiting the number of basic constructs and keywords. Having these constructs and keywords close to what one would find in natural language enhances the easeness of writing specifications or programs.

Modularity

Modularity means that the language allows for designing modules that can be reused as parts of other specifications. Modularity can be seen as abstraction where complicated structures can be defined and reused later with hiding or ignoring the details.

Expressiveness

Expressiveness means that the language has operations and functions that allow for articulating the specification of product families. The language should allow for the expression of all the notions of dependencies among product families that we find in the literature. It should also express them in a simple and elegant way.

Deterministic Interpretation

A specification has a deterministic interpretation if it performs as expected under all conditions. We expect that the proposed language will have one interpretation for each of the models of product family algebra (i.e., set and bag model).

3.3 Design and Structure

In this section, I discuss the design and structure of the proposed language. I present the grammar of the proposed language and the structure of a specification written in the language. Then, I discuss the semantics of each of the language constructs.

3.3.1 Syntax

```
1  include file.spec
2  bf gps
3  bf camera
4  bf mp3_player
5  bf basic_screen
6  bf touch_screen
7  bf keypad
8  bf calling_feature
9  accessories = some_of ( gps, camera, mp3_player)
10 screen = one_of ( basic_screen , touch_screen)
11 mobile = all_of ( opt(accessories), calling_feature , opt(keypad),
       screen)
```

Listing 3.1: Mobile Phone Product Family Specification

The specification of Listing 3.1 shows two main components of the specification structure: basic feature introductions and family definitions. In line 1, the keyword "include" tells the compiler to include the content of the referenced file into the current specification. Lines 2 – 8 define the basic features *GPS*, *Camera*, *MP3_Player*, *Basic_Screen*, *Touch_Screen*, *Keypad*, and *Calling_feature*, respectively. Lines 9 – 11 define the families *accessories*, *screen*, and *mobile*.

Figure 3.2 shows the grammar of the proposed language. Every specification ⟨spec⟩ consists of: inclusion section ⟨inclusion⟩, basic feature declaration ⟨declaration⟩, family defenition section ⟨body⟩, constraints section ⟨constraints⟩, and conditions section ⟨if_statements⟩. These sections have to follow the same order in a specification. Some

sections are mandatory and others are optionals. In the following, we consider each one of the sections.

⟨spec⟩ ⟶ ϵ | ⟨spec⟩⟨inclusion⟩⟨declaration⟩⟨body⟩⟨constrints⟩⟨if_statements⟩
⟨inclusion⟩ ⟶ ϵ | ⟨inclusion⟩“include”⟨identifier⟩
⟨declaration⟩ ⟶ “bf”⟨identifier⟩ | ⟨declararion⟩“bf”⟨identifier⟩
⟨identifier⟩ ⟶ id
⟨body⟩ ⟶ ⟨labelled_family⟩ | ⟨body⟩⟨labelled_family⟩
⟨labelled_family⟩ ⟶ ⟨idenntifier⟩“=”⟨term⟩
⟨term⟩ ⟶ ⟨identifier⟩ | number | ⟨term⟩“+”⟨term⟩ | ⟨term⟩“ · ”⟨term⟩
| (⟨term⟩) | “all_of”(⟨family_list⟩) | “one_of”(⟨family_list⟩)
| “some_of”(⟨family_list⟩) | number (⟨family_list⟩)
| number .. number (⟨family_list⟩) | ⟨term⟩^ number
| ⟨term⟩^ number .. number | “opt”(⟨term⟩)
⟨family_list⟩ ⟶ ⟨term⟩ | ⟨term⟩“,”⟨family_list⟩
⟨constraints⟩ ⟶ ϵ | ⟨constraints⟩⟨constraint⟩
⟨constraint⟩ ⟶ “in”⟨identifier⟩“,”⟨term⟩“require”⟨term⟩
| “in”⟨identifier⟩“,”⟨term⟩“exclude”⟨term⟩
| “in”⟨identifier⟩“,” “every family has”⟨term⟩
| “in”⟨identifier⟩“,” “most occurrence of”⟨term⟩“is”number
| “globally” “,”⟨term⟩“require”⟨term⟩
| “globally” “,”⟨term⟩“exclude”⟨term⟩
⟨if_statements⟩ ⟶ ϵ | ⟨if_statements⟩⟨if_statement⟩
⟨if_statement⟩ ⟶ “if”⟨expr⟩then⟨statement⟩
| “if”⟨expr⟩“then”⟨statement⟩“else”⟨statement⟩
⟨expr⟩ ⟶ ⟨identifier⟩“equal”⟨identifier⟩ | ⟨identifier⟩“subfamily”⟨identifier⟩
| ⟨identifier⟩“refine”⟨identifier⟩ | “size”⟨identifier⟩⟨relator⟩⟨a_number⟩
| ⟨expr⟩“and”⟨expr⟩ | ⟨expr⟩“or”⟨expr⟩
| “not”⟨expr⟩ | (⟨expr⟩)
⟨a_number⟩ ⟶ number | “size”⟨identifier⟩
⟨relator⟩ ⟶ “==” | “ > ” | “ >= ” | “ < ” | “ <= ”
⟨statement⟩ ⟶ ⟨constraints⟩ | (⟨if_statement⟩)

Figure 3.2: Language Grammar

Inclusion Section

The inclusion section is the first part of a specification. It is an optional part which consists of one or more inclusion lines. Each line starts with the clause *include* which allows for reuse of already written specifications by importing the referenced file into the current specification. The format of an inclusion line is

include ⟨file_name⟩

Basic Feature Declaration Section

The basic feature declaration section introduces a sequence of individual basic features of the form

bf ⟨feature_name⟩

An example of basic features are the features *GPS* and *Camera* in the mobile phone product family given in Listing 3.1.

Family Definition Section

The family definition section is a sequence of one or more family definitions. Each family definition starts with the family name, followed by " = ", followed by a term.

⟨family_name⟩ = ⟨term⟩

A term is an algebraic expression that is either a basic term or a compound term. A basic term is a feature which is an already defined basic feature or family. A compound term is defined using one or more features and a relation connecting them. A relation is one of the following:

- $opt(f)$: It indicates that the feature is optional.
- $all_of(\,f1, f2, \ldots)$: It means that all features are selected in the family.
- $one_of(\,f1, f2, \ldots)$: It means that one feature is selected in the family.
- $some_of(\,f1, f2, \ldots)$: It means that one or more (up to the actual number of features in the list) features are selected in the family.
- $n\,(\,f1, f2, \ldots)$: it means that n features are selected in the family.
- $n..m\,(\,f1, f2, \ldots)$: It means at least n features or at maximum m features are selected in the family.
- $f1 + f2$: It indicates the choice between the two features.
- $f1 \cdot f2$: It indicates the composition of features.

Feature cardinality is defined in one of the two ways:

- $f\,\hat{}\,n$: It specifies that the feature is repeated n times.
- $f\,\hat{}\,n..m$: It specifies that the feature is repeated between n and m times.

Constraints Section

The constraints section of a specification is a list of constraints. Constraints are used to eliminate undesired products from a certain family or all families. A constraint

has one of the following forms:

$$\textbf{in}\ \langle\text{family_name}\rangle,\ \langle\text{family_name}\rangle\ \textbf{require}\ \langle\text{family_name}\rangle \tag{3.1}$$

$$\textbf{in}\ \langle\text{family_name}\rangle,\ \langle\text{family_name}\rangle\ \textbf{exclude}\ \langle\text{family_name}\rangle \tag{3.2}$$

$$\textbf{in}\ \langle\text{family_name}\rangle,\ \textbf{every family has}\ \langle\text{family_name}\rangle \tag{3.3}$$

$$\textbf{in}\ \langle\text{family_name}\rangle,\ \textbf{max occurrence of}\ \langle\text{feature_name}\rangle\ \textbf{is}\ \langle\text{number}\rangle \tag{3.4}$$

$$\textbf{globally},\ \langle\text{family_name}\rangle\ \textbf{require}\ \langle\text{family_name}\rangle \tag{3.5}$$

$$\textbf{globally},\ \langle\text{family_name}\rangle\ \textbf{exclude}\ \langle\text{family_name}\rangle \tag{3.6}$$

Constraints 3.1 and 3.2 mean that in *family_name* the existence of the first *family_name* requires or excludes the existence of the second *family_name*, respectively. Constraint 3.3 indicates that every product of a specified family must have a specified family. To limit the number of occurrences of certain feature in a family, we use a Constraint 3.4. Constraints 3.5 and 3.6 mean that in all families in the specification, the existence of the first *family_name* *requires* or *excludes* the existence of the other *family_name*, respectively.

Conditions Section

The conditions section of the structure is a sequence of conditions. A condition is an *if* statement that consists of a condition, a *then* part, and an optional *else* part.

if ⟨condition⟩ **then** ⟨constraints⟩

if ⟨condition⟩ **then** ⟨constraints⟩ **else** ⟨constraints⟩

A condition is one or more constraints with a guard. If the guard evaluates to true then the constraints in the *then* part are executed, otherwise the constraints in the *else* part are executed, if they exist. A basic guard can be one of the following forms:

⟨family_name⟩ **equal** ⟨family_name⟩

⟨family_name⟩ **subfamily** ⟨family_name⟩

⟨family_name⟩ **refine** ⟨family_name⟩

size ⟨family_name⟩ **relator** ⟨number⟩

The *relator* is an arithmetic relator and it can be one of >, >=, <, <=, or ==. A compound guard is defined using basic guards connected with AND, OR, and NOT.

Comments

The language accepts comments to add information to the specification that will not be compiled. A comment is a single line comment that starts with double backslashes "\\". The percentage sign "%", which is the comment of the language of Jory, is also acceptable.

3.3.2 Semantics

In Section 2.2, I have presented the PFA models (i.e., set and bag). Also, In Section 3.3.1, I have presented the different constructs of the proposed language. In this section, I present the semantics of the proposed language by showing how each construct is translated to PFA.

The optional feature $opt(f1)$ is translated to the choice between the feature $f1$ and the family of one product that has no features (i.e., $f1 + 1$). The operations $\cdot$ and $+$ stay the same as in PFA. The construct $all_of()$ translates to the composition of the features. On the other hand, $one_of()$ translates to the choice between the list of features. However, $some_of()$ translates to the choice between every possible composition of features. $n()$ translates to the choice between every composition of n features. Similarly, $n\,..\,m()$ is the choice between the composition of n to m features. Feature cardinality $\hat{}n$ translates to the composition of the feature n times. The cardinality $\hat{}n\,..\,m$ translates to the choice between the composition of the feature n times to m times.

The language constraint $\textbf{in}\, a, b\, \textbf{require}\, c$ translates to $b \xrightarrow{a} c$ (i.e., if a has b then a has c). However, the constraint $\textbf{in}\, a, b\, \textbf{exclude}\, c$ translates to $b \cdot c \xrightarrow{a} 0$ (i.e., the product of family a that has b and c is invalid). The constraint $\textbf{in}\, a,$ **every family has** b translates to $1 \xrightarrow{a} b$ (i.e., every product of a has b). Finally, the constraint $\textbf{in}\, a,$ **max occurrence of** b **is** n indicates that every product that contains more that n features is an invalid product. Therefore, it can be expressed as $b^{n+1} \xrightarrow{a} 0$.

Table 3.2 summaries the language semantics by showing each construct and its equivalent PFA term.

Proposed Language	PFA
$\mathbf{opt}(f)$	$f+1$
$f_1 \cdot f_2$	$f_1 \cdot f_2$
$f_1 + f_2$	$f_1 + f_2$
$\mathbf{all_of}(f_1, \cdots, f_n)$	$\cdot(i \mid 1 \le i \le n : f_i)$
$\mathbf{one_of}(f_1, \cdots, f_n)$	$+(i \mid 1 \le i \le n : f_i)$
$\mathbf{some_of}(f_1, \cdots, f_n)$	$F \triangleq \cdot(i \mid 1 \le i \le n : (1+f_i))$ and we add the following constraint $1 \xrightarrow{F} \mathbf{one_of}(f_1, \cdots, f_n)$
$\mathbf{m}\,(f_1, \cdots, f_n)$ with $m \le n$	$F \triangleq \cdot(i \mid 1 \le i \le n : (1+f_i))$ and we require that $\forall(p \mid p \in F \wedge p \text{ is a product} : p \in \mathbb{F}^m)$, where $\mathbb{F} = \{f_1, \cdots, f_n\}$
$\mathbf{m\ ..\ l}\,(f_1, \cdots, f_n)$	$F \triangleq \cdot(i \mid 1 \le i \le n : (1+f_i))$ and we require that $\forall(p \mid p \in F \wedge p \text{ is a product} : p \in \cup(j \mid m \le j \le l : \mathbb{F}^j))$, where $\mathbb{F} = \{f_1, \cdots, f_n\}$
$f\,\hat{}\,\mathbf{m}$	$\cdot(i \mid 1 \le i \le m : f)$
$f\,\hat{}\,\mathbf{m\ ..\ l}$	$+(j \mid m \le j \le l : \cdot(i \mid 1 \le i \le j : f))$
in a, f_1 **require** f_2	$f_1 \xrightarrow{a} f_2$
in a, f_1 **exclude** f_2	$f_1 \cdot f_2 \xrightarrow{a} 0$
in a, **every family has** f	$1 \xrightarrow{a} f$
in a, **max occurrence of** f **is** m	$\cdot(i \mid 1 \le i \le m+1 : f) \xrightarrow{a} 0$

Table 3.2: Proposed Language Semantics

3.4 Assessment

In this section we assess the proposed language. The assessment is done by performing a comparison study between the feature modelling techniques listed in Figure 3.3 and our language. The criteria of the comparison are readability, writability, modularity, and expressiveness.

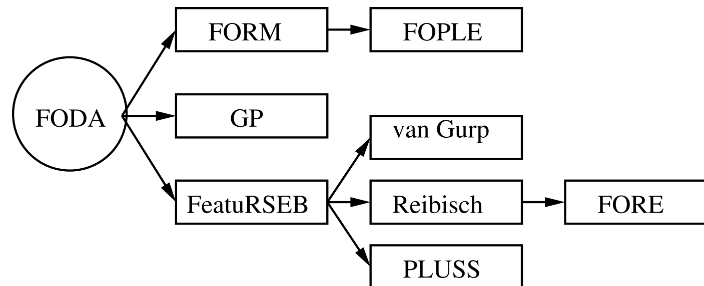

Figure 3.3: FODA Extension Feature Modelling Techniques

Readability

We compare the proposed language to the graphical feature modelling techniques given in Figure 3.3 in terms of ease of reading and clarity of meaning.

The techniques *FODA*, *FORM*, and *GP* have a limited number of constructs and their constraints are textual. Their models are easy to read. Since graphical feature models, when small and compact, are easy to visualize and comprehend. However, modelling a large system generates a large feature model which is tedious to read and comprehend. The model in *FOPLE* consists of four layers and the three types of feature relations make it less readable. Although, *FeatuRSEB*, *van Gurp*, *Riebusch et al.*, and *PLUSS* have clear and limited constructs, these techniques model constraints graphically. If there are many constraints in the model, they can clutter the model and make it less readable. The proposed language is easy to read because it has a limited and comprehensive constructs and its constraints are of the same nature of the model (i.e., textual). Moreover, the proposed language constructs are close to natural language which makes them easy to read and guess their meaning by novice users. On the other hand, the constructs in graphical feature models need a prior knowledge of their meanings.

Writability

Writability and readability of a feature modelling technique are directly related and affected by the same properties. Therefore, what applies previously for readability applies as well for writability. Therefore, models in *FODA*, *FORM*, and *GP* are easy to write for their limited number of constructs, while models in *FOPLE* are not writable. Constraints in *FeatuRSEB*, *van Gurp*, *Riebusch et al.* , and *PLUSS* models are easier to write than to read. So their models are writable. Moreover, writing specifications in the proposed language is easy for the same reasons as readability.

Modularity

All the feature modelling techniques in Figure 3.3 do not support modularity. In our language, modularity is supported by using the “include” keyword to include a pre-written specification in the current specification.

Expressiveness

In Appendix A, Table A.1 shows the constructs of each technique in Figure 3.3 as well as our language. Moreover, it shows the equivalence between these techniques. The table shows clearly that our language can express all of the constructs of the other techniques.

Study Summary

To summarize, Table 3.3 shows the summary of the study. It shows whether a technique has a full support, restricted support, or no support for the considered criteria.

	Readability	**Writability**	**Expressiveness**	**Modularity**
FODA	full support	full support	restricted support	no support
FORM	full support	full support	restricted support	no support
FOPLE	no support	no support	restricted support	no support
GP	full support	full support	full support	no support
FeatuRSEB	restricted support	full support	full support	no support
van Gurp	restricted support	full support	full support	no support
Reibisch	restricted support	full support	full support	no support
PLUSS	restricted support	full support	full support	no support
FORE	restricted support	full support	full support	no support
Proposed Language	full support	full support	full support	full support

Table 3.3: Study Summary

3.5 Conclusion

In this chapter, we considered the design of the proposed language. The structure of the language is composed of five sections: inclusion section, basic feature declaration section, family definition section, constraints section, and conditions section. I showed the actual syntax and structure of the language by presenting the grammar of the language and syntax of each section. Moreover, I presented the semantics of the language by showing the equivalent PFA for each construct. Finally, I assessed the language with regard to the criteria of readability, writability, modularity, and expressiveness by performing a comparison study to other feature modelling techniques found in the literature.

Chapter 4

Design of the Compiler as a Part of the Tool Jory

In this chapter, we consider the design and implementation of a compiler for the proposed language. Our aim is to make the compiler a part of an integrated environment for writing and executing specifications. My work is an extension of the tool Jory. We also extend Jory with a user interface to accommodate the addition of the developed compiler. The compiler translates the high level language presented in Chapter 3 to the low level language of PFA (i.e., to that of an idempotent semiring). Then, the tool Jory runs PFA code. The tool is designed in a layered architecture style.

To better understand the design of the compiler, we discuss the design of compilers and interpreters in general in Section 4.1. Afterwards, I present the detailed design of the compiler for the proposed language in Section 4.2.

4.1 Compiler Structure

A compiler is a system that among its functionalities is the translation of programs written in one language to their equivalent programs in another language [ALSU06]. Moreover, a compiler detects and reports errors in the source program. Figure 4.1 shows the general view of a compiler. The difference between a compiler and an interpreter is that the latter executes the output on the machine directly. The compiler generates the output without executing it. With this understanding, we can say that the developed compiler is one in which its generated code is executed on a virtual machine (i.e., Jory).

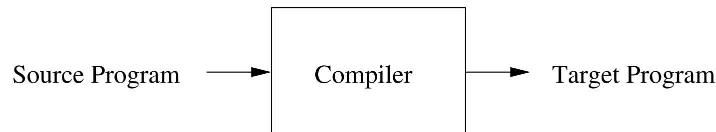

Figure 4.1: A General View of a Compiler

Looking into the details of a compiler, we find that the main two components of a compiler structure are the *front-end* (analysis) and the *back-end* (synthesis) as shown in Figure 4.2.

The front-end reads the source program, checks its grammatical structure, reports any detected errors, then constructs, internally, an intermediate representation equivalent to the source program. Moreover, the front-end collects information about the program in a symbol table. The back-end takes the intermediate representation code and generates the equivalent target program. A symbol table is a data structure that holds the names of variables of the source program and the information about them.

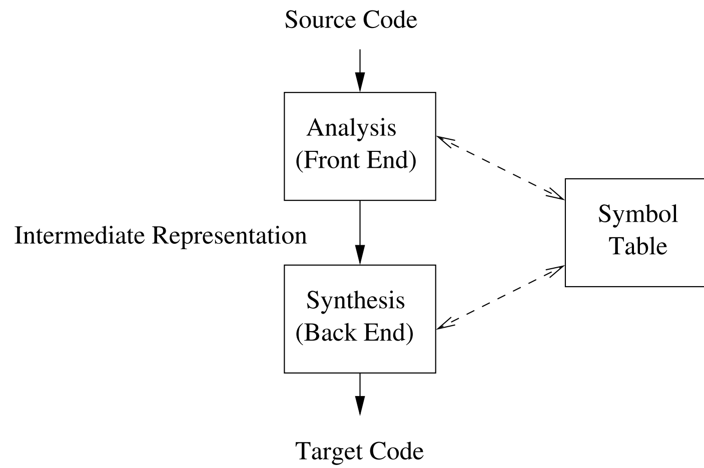

Figure 4.2: The Main Components of a Compiler

This information includes the type, scope, and other information needed to generate the target code.

The compiler structure consists of the phases shown in Figure 4.3. Instead of a staged execution of each one of the phases, a sequence of them are combined to be executed together. For instance, the front-end phases are grouped together in one execution and the back-end phases are combined into another execution.

The lexical analyzer reads the character stream of the source code and groups the characters into lexemes. For each lexeme, the lexical analyzer passes the token type along with the lexeme value to the syntax analyser.

The syntax analyzer or parser uses the tokens passes by the lexical analyzer to check the grammatical structure of the source program. It generates an intermediate representation of the source program. The intermediate representation has many forms

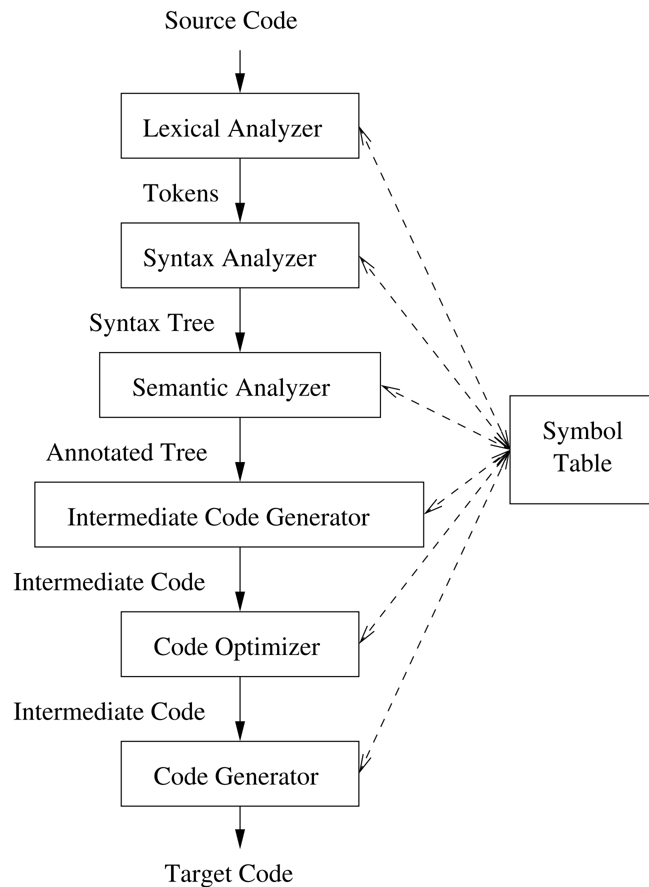

Figure 4.3: The Compiler Phases

but it is most commonly a tree called a parse tree. In the case of a grammatical error, it reports the error to the user.

The semantic analyzer uses the parse tree and the symbol table to check the semantic consistency of the source program. Moreover, the semantic analyzer collects type information to be used later for code generation. One of the most important roles of the semantic analyzer is type checking. The semantic analyzer checks the consistency of types, especially in arithmetic operators, and performs the needed type conversion.

In the process of compilation, the compiler generates multiple intermediate representations between phases. One important intermediate representation in most compilers is intermediate code. Intermediate code is generated between syntax and semantic analysis phases, or between front-end and back-end. The intermediate code has to be easy to construct from the source program and easy to translate into the target program. The main role of the code optimizer is to improve the quality of the intermediate code. It improves the quality of the code by eliminating redundant and unnecessary code. The code optimizer combines multiple statements and shortens the code. Finally, the code generator phase takes the intermediate code, processes it, and generates the target code.

4.1.1 Flex and Bison

To construct the front-end of the compiler, I used *Flex* and *Bison* [Lev09], which are tools that perform lexical analysis and syntax analysis, respectively.

Flex is a tool to build lexers. It is used to look for patterns (lexemes) described in regular expressions and performs actions. If *Flex* is used with *Bison*, then the action would be to pass the appropriate tokens and lexemes to the parser. A *Flex* program is mostly C code. It consists of three sections separated by %% lines. The fist section contains definitions and settings. The second section contains regular expressions and their actions. The third section contains C code that provides functions and procedures that are copied to the generated lexer.

In the first section, the code between "%{" and "%}" is copied to the generated lexer. In the second section, each regular expression starts at the beginning of the line followed by the action between "{" "}". The action is one or more C statements. Listing 4.1 is a *Flex* lexer for the calculator example given Figure 3.1 in Chapter 3.

```
%%
"+"    {return PLUS;}
"*"    {return MUL;}
"("    {return OP;}
")"    {return CP;}
[0-9]+  {yylval = atoi(yytext); return NUMBER;}
\n      {return NL;}
[ \t]     %ignore white space%
.         %ignore anything else%
%%
```

Listing 4.1: *Flex* Lexer for Calculator

In practice, a *Flex* lexer works with a parser, which is the main program. Whenever the parser needs a token, it calls the lexer to return the next token. Every time the lexer gets called, it remembers where it was on the input stream, and resumes on the next call.

Bison is a tool used to build parsers. The parser's job is to identify the relationship between the tokens. A *Bison* parser contains the same parts as a *Flex* lexer. The first section is the declaration section. The C code between "%{" and "%}" is copied to the generated parser. The C code is followed by token declarations. The second section is a BNF grammar. The grammar section is a set of rules. Each rule is followed by an action between "{" "}" telling the parser what to do when a rule is matched.

The value of the left hand side is indicated by “$$” while the values of the right hand side are “$1”, “$2”, and so on. The value of a token is the value assigned to *yylval* in the lexer.

Listing 4.2 is the *Bison* parser for the calculator example.

```
%{
#include <stdio.h>
%}
%token PLUS MUL OP CP
%token NUMBER NL
%%
program: expr NL { printf(" = %d\n", $1);}
;
expre: expr PLUS term {$$ = $1 + $3;}
  | term
;
term: term MUL factor {$$ = $1 * $3;}
  | factor
;
factor: OP expr CP { $$ = $2;}
  | NUMBER
;
%%
main()
{
  yyparse();
}
```

Listing 4.2: *Bison* Parser for Calculator

4.2 System Design

In this section, I present the design of the compiler. First, I give an overview of the design and the layers of the tool. Afterwards, I show the detailed design of the new layers in Jory.

4.2.1 Overview

System design is the presentation of the major components of the system and the communication between them [Som01]. There are different types of architectures for systems. We adopt a layered architecture to conceptually decompose our parser. We choose a layered architecture due to the nature of compiler structure and the design of the tool Jory.

The layers of Jory are the *User Interface Layer*, the *Syntax Analysis Layer* or the front-end of the compiler, the *Code Generation Layer* or the back-end of the compiler, the *Term Evaluation Layer*, the *Concrete Model Layer*, and the *BDD Layer*. Figure 4.4 shows the layers of Jory. The *User Interface Layer* is implemented in Python while the *Syntax Analysis Layer* is implemented in *Flex* and *Bison* which generates C code. The *Code Generation Layer* and the *Term Evaluation Layer* are implemented in Haskell. The *Concrete Model Layer* and the *BDD Layer* are implemented in C/C++. The first three layers are the subject of this thesis.

The *User Interface Layer* is the interface between the user and the tool's internal components. The user writes a specification in the proposed language and chooses the model in which the computation ought to be carried (i.e., set or bag). Also, the

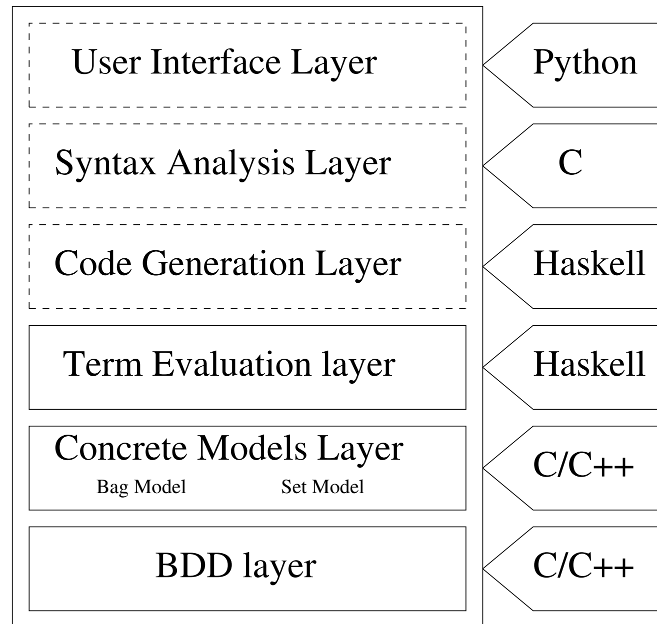

Figure 4.4: Layers of Jory Tool

user compiles the specification and performs queries on the specification. Compilation messages and the output of executing the specification are presented on the interface. The *Syntax Analysis Layer* reads the specification, checks the lexical and grammatical structure, reports any syntax errors, and generates the intermediate code for the back-end. The *Code Generation Layer* takes the intermediate code, processes it and generates the equivalent PFA code. The *Term Evaluation Layer* reads PFA code, evaluates the basic features and families, and generates a registry for them. The *Concrete Model Layer* uses the registry from the previous layer to generates BDD code in the selected model. Finally, the *BDD Layer* executes the BDD code and generates the output.

The communication between the layers is top-down, with the *User Interface Layer* as a controller of the layers.

4.2.2 Detailed Design

In this section, we consider the detail design of the three layers added to Jory: the *User Interface Layer*, the *Syntax Analysis Layer*, and the *Code Generation Layer*. The detailed design of the other layers can be found in [Alt10].

User Interface Layer

The interface that I propose, shown in Figure 4.5, has a text editor allowing the user to write a specification or to open an already written one. Moreover, it has two tabs for showing compilation errors and the output. Also, it has a radio buttons for the user to select the model which the specification will be compiled in, and the maximum occurrences in the case of a bag model. There is a part to write queries on the specification and another part for aspects. The queries part is used to select operations to be executed on the compiled specification.

The *User Interface Layer* is written in Python and consists of three classes: *main*, *compiler controller*, and *executer*.

- **main**

  Secret: It is a class that contains the main components of the frame.

  Service: It initializes the interface components. Moreover, it hides the events handlers of the interface components (e.g., radio button change and button press).

- **compiler controller**

  Secret: It is a class that contains the methods controlling the compiler.

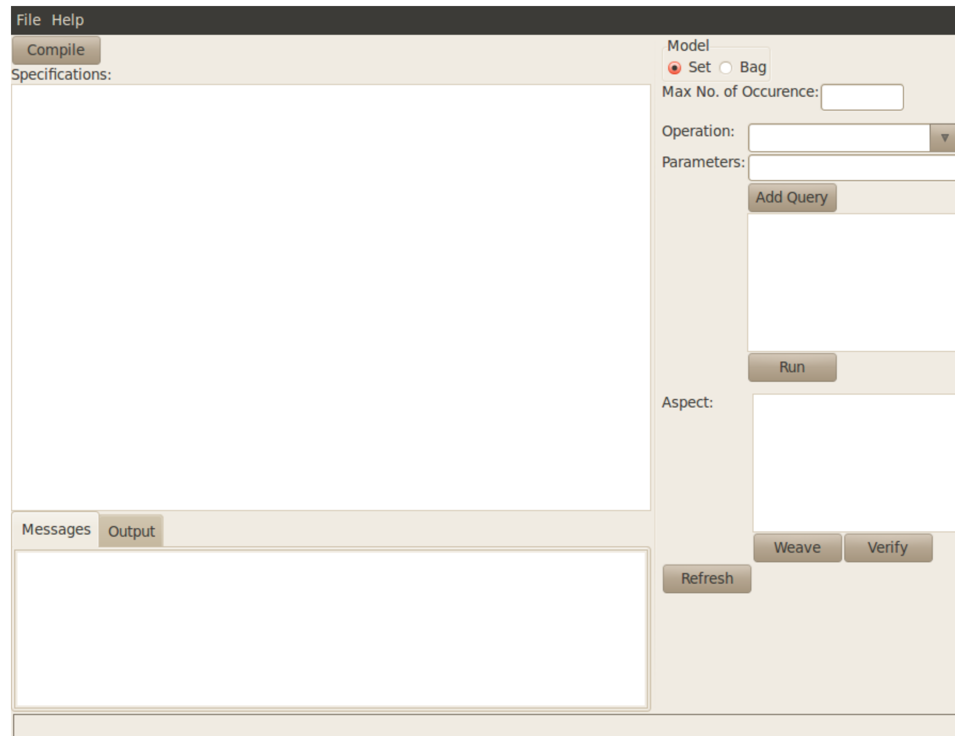

Figure 4.5: User Interface

Service: It encompasses the methods that call and process the layers of the *Syntax Analysis Layer*, the *Code Generation Layer*, the *Term Evaluation Layer*, and the *Concrete Model Layer*.

- **executer**

  Secret: It is a class that contains the methods controlling the execution of the *BDD Layer*.

  Service: It holds the methods that process queries, compiles and executes BDD code, and shows the results to the user.

Syntax Analysis Layer

The *Syntax Analysis Layer* takes the specification written in the proposed language, checks the lexical and syntactical structure of the specification, reports compilation errors, and generates intermediate code. This layer consists of a lexical analyzer (lexer), a syntax analyzer (parser), a symbol table, and a module to handle parse trees as shown in Figure 4.6.

- **lexical analyzer**

  Secret: It is an algorithm that does lexical analysis of the specification.

  Service: It reads the stream character of the source code, identifies lexemes, and returns token types and lexemes to the parser. The lexer is written in *Flex*.

- **syntax analyzer**

  Secret: It is an algorithm that checks the grammatical structure of the specification.

  Service: It takes the tokens generated by the lexer and organizes them according to the grammatical structure of the language. It also reports syntax errors and builds internal partial parse trees for families, constraints, and conditions. Finally, the parser generates the intermediate code from these trees. The parser is written in *Bison*.

- **symbol table**

  Secret: It is a data structure that stores identifiers.

  Service: It is used to store information about identifiers (basic features and

families). This information is used later to check that an identifier is not repeated in the specification and that an identifier is defined before it is used.

- **tree handler**

  Secret: It is a data structure that stores parse trees.

  Service: It is used to build partial parse trees and generate intermediate code from them.

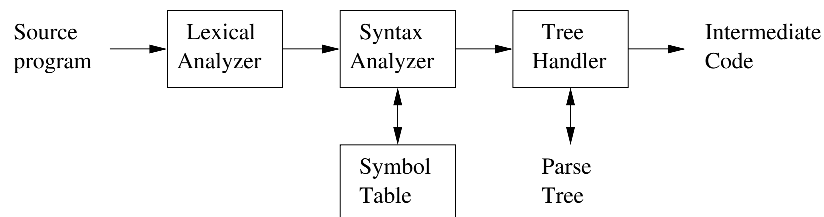

Figure 4.6: A Model of the *Syntax Analysis Layer*

Code Generation Layer

The *Code Generation Layer* takes the intermediate code generated by the *Syntax Analysis Layer*, processes it, and generates PFA code for feature and families, and C++ code for constraints and conditions.

This layer consists of one main module and four submodules. The main module is code generator and the submodules are respectively for basic features, families, constraints, and conditions as shown in Figure 4.7.

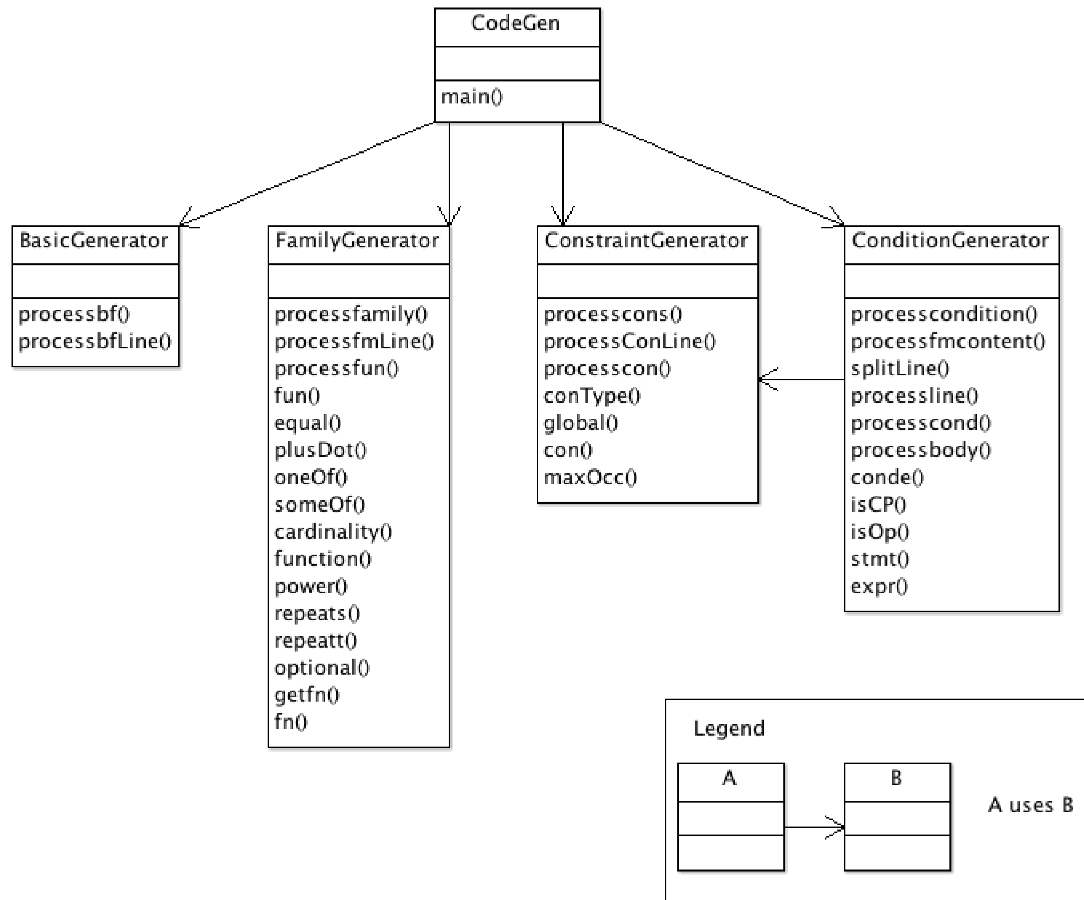

Figure 4.7: Module Uses Diagram of the Code Generator

- **Basic Generator**

  Secret: It is an algorithm that translates intermediate code for basic features into PFA.

  Service: This module hides the functions that takes the intermediate code for basic features and generates the equivalent PFA code.

- **Family Generator**

  Secret: It is an algorithm that generates PFA code from intermediate code for product family definitions.

  Service: This module hides the functions that read the family section of the intermediate code and processes each family definition separately. Then, it translates each family definition recursively to PFA code.

- **Constraint Generator**

  Secret: It is an algorithm that translates the intermediate code for constraints to PFA.

  Service: This module hides the functions that read the intermediate code for constraints and translate it to C++ code for the *BDD Layer* to execute the specification.

- **Condition Generator**

  Secret: It is an algorithm that translates conditions from their intermediate representation to their equivalence in PFA.

  Service: This module hides the functions that read the condition section of the intermediate code, translate the if conditions to C++, and call *Constraint Generator* to translate the constraints.

Let us consider a sample of the Code Generator code shown in Listing 4.3. The *CodeGen* module has the following function "*main*":

```
1  main = do args <- getArgs
2         f <- loadFile (args!!0)
3         processbf  f (args !!4)
4         f <- loadFile (args !!1)
5         processfamily f (args !!4)
6         f <- loadFile (args !!2)
7         processcons f (args !!5)
8         f <- loadFile (args !!3)
9         processcondition f (args !!5)
```

Listing 4.3: Code of *CodeGen* Module

The function *main* loads the arguments and calls the appropriate module. In line 4, it loads the file that contains the intermediate code for the family section. Then in line 5, it calls the function *processfamily* in the module *FamilyGenerator*. The module *FamilyGenerator* contains the functions that process the family section and generate the PFA code for that section. The code of *FamilyGenerator* is presented in Appendix B.

Illustration

Figure 4.8 shows an illustration of a line of the mobile specification. The line is passed from the *User Interface Layer* to the *Syntax Analysis Layer* where the parser generates an internal parse tree. Then, the intermediate code is generated from the parse tree, which is a prefix notation of the term. Afterwards, the intermediate code is processed in the *Code Generation Layer* and the equivalent PFA code is generated.

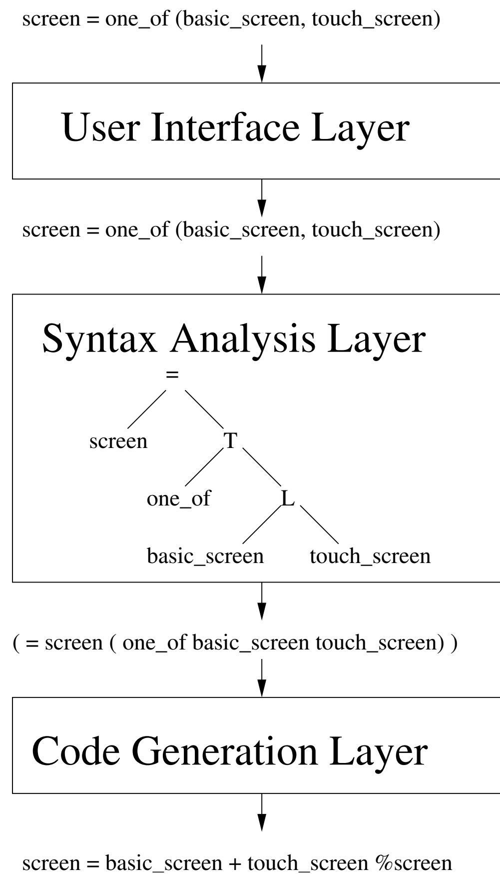

Figure 4.8: Tool Illustration

4.3 Conclusion

In this chapter, I presented the design of a PFA compiler as an extension of the tool Jory. I presented the design by giving a brief background on the design and structure of compilers. Moreover, I gave an overview on *Flex* and *Bison* tools. Finally, I presented the detailed design of our compiler.

The architecture design of our compiler is a layered architecture. The layered architecture is the most appropriate design for a number of reasons. First, it is the design

of most compilers. Second, it is the design of the tool Jory which our compiler is a part of. Moreover, the layered architecture is modifiable and portable. The layered architecture is the most effective design for changes and maintenance. If a layer is replaced or changed internally without changing the output of the layer, this does not affect the system or other layers. For example, if we want to change the language grammar, we can change the parser code in the *Syntax Analysis Layer* without affecting the other layers of the tool. However, if the output of a layer is changed, then we only need to change the adjacent layer accordingly. For instance, if we would like to add a new construct to the language, then we would need to update the *Syntax Analysis Layer*. This will change the intermediate code that is output from this layer and we would need to update *Code Generation Layer* to handle the change of its input.

Chapter 5

Case Study

In this chapter, I present a case study of an e-shop system. I use the language and the tool on a case study that is a reference within the feature modelling community [Lau06]. The aim is to show that the language and the tool are suitable for large real-world applications. In Section 5.1, I introduce the e-shop system. In Section 5.2, I show the specification of the system written in the proposed language. In Section 5.3, I present the results after running the specification on the tool Jory. Finally, in Section 5.4, I conclude the case study.

5.1 E-shop System

An e-shop system is a web site that allows companies to do business over the internet. A company uses an e-shop system to sell products and/or services to customers. The system deals with multiple customers, products and services, and stakeholders aiming to deliver the products or services to the customer.

The e-shop system that I am presenting is inspired from [Lau06]. For more details about the system, I refer the reader to [Lau06]. Due to the size of the system, it will be presented in subfamilies with a FODA feature model.

The e-shop system is constructed from two main subfamilies *store_front* and *business_management*, as shown in Figure 5.1. The system consists of almost 290 features and 21 constraints.

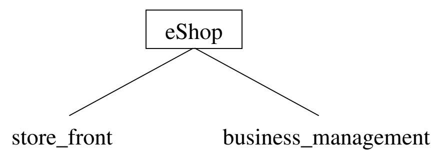

Figure 5.1: E-shop System Overview

5.1.1 Store Front

The *store_front* is the interface of the e-shop system. It is a subfamily containing the features that are related to the web site interface, which impacts the user experience directly. Figure 5.2 shows the *store_front* subfamily. The subfamily consists of two mandatory subfamilies *catalog* and *buy_paths*. Moreover, it consists of five optional subfamilies *home_page*, *registration*, *wish_list*, *customer_service*, and *user_behaviour_tracking*.

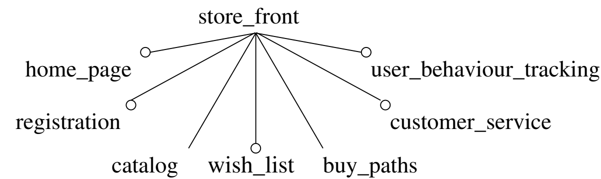

Figure 5.2: Store Front Overview

Home Page

The *home_page* is the first page of the e-shop that a user encounters when using the website. Figure 5.3 shows the *home_page* subfamilies and features. The content of *home_page* can be generated dynamically, statically, or both.

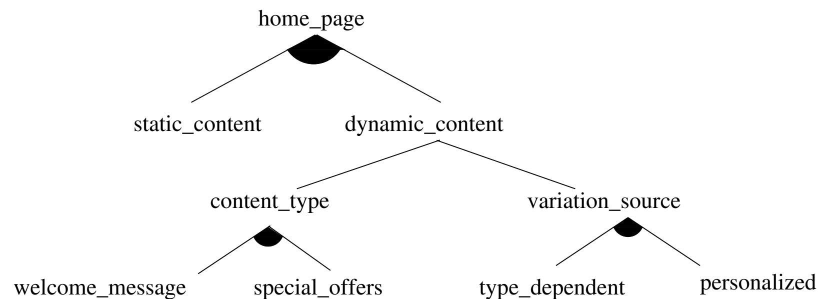

Figure 5.3: Home Page Overview

Registration

The *registration* is an optional subfamily. It allows for storing user information. It is useful for the customers so they do not need to reenter their information with every purchase. Moreover, it enables the company to develop better targeting strategies. Figure 5.4 shows the subfamilies of *registration*.

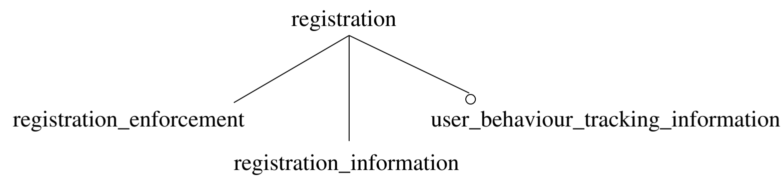

Figure 5.4: Registration Overview

Catalog

The *catalog* is where products and services of the e-shop system are presented to the customer. Therefore, it is a mandatory to have in the system. The *catalog* is a framework that organizes products and services which affect the customers shopping experience. Figure 5.5 shows the *catalog* subfamily and its children.

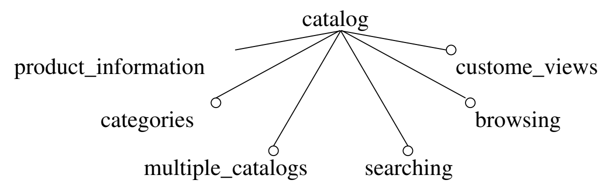

Figure 5.5: Catalog Overview

Wish List

The *wish_list* is where the customer can store a list of products that they wish to buy or receive as gifts. Also, it allows the customer to keep track of the list prices. Figure 5.6 shows the *wish_list* subfamily and its subfamilies.

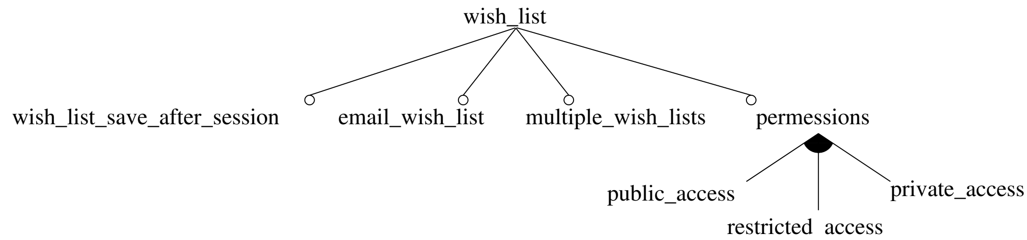

Figure 5.6: Wish List Overview

Buy Paths

The *buy_paths* is a collection of customer purchase workflow features. It includes *shopping_cart*, *checkout*, and *order_confirmation*. Figure 5.7 offers a view on the *buy_paths* subfamily.

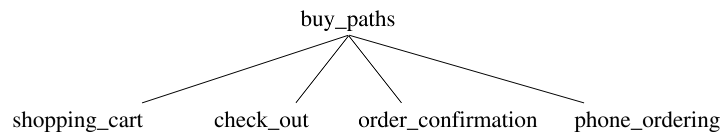

Figure 5.7: Buy Paths Overview

Customer Service

The *customer_service* is a subfamily that contains features and subfamilies that deals with customer services and shopping experience feedback. Figure 5.8 shows the subfamily and its children.

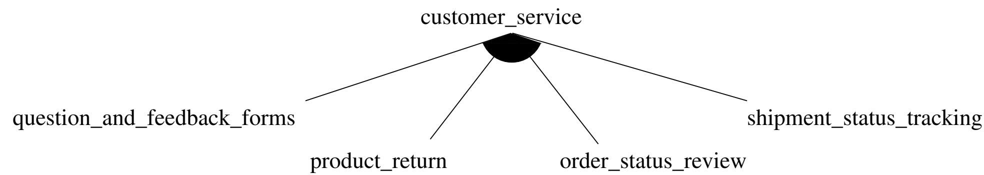

Figure 5.8: Customer Service Overview

User Behaviour Tracking

The *user_behaviour_tracking* subfamily has the information needed to monitor and track user behaviours. It helps the company to develop better marketing strategies and to generate studies on customer trends. Figure 5.9 shows the subfamily *user_behaviour_tracking* and its children.

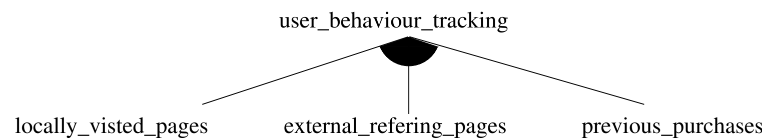

Figure 5.9: User Behaviour Tracking Overview

5.1.2 Business Management

The *business_management* subfamily is a group of features that work as the back-end of the system. It is used by the e-shop staff and management. Figure 5.10 shows the subfamilies of this group.

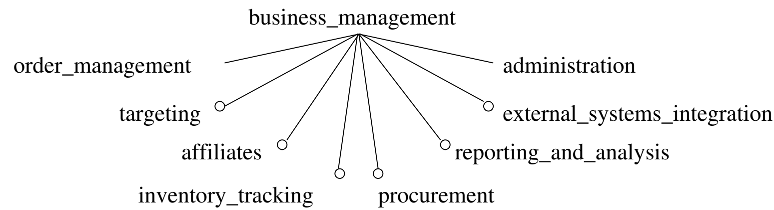

Figure 5.10: Business Management Overview

Order Management

This subfamily contains the features that deal with order fulfillment. Figure 5.11 shows the *order_management* subfamily and its subfamilies.

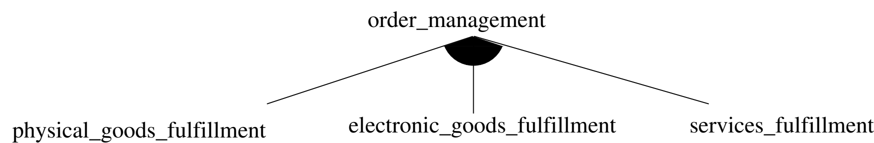

Figure 5.11: Order Management Overview

Targeting

The *targeting* subfamily contains all the features and subfamilies that deal with marketing and promotion of products and services. It helps to improve e-shop sales and company revenues. Figure 5.12 presents the *targeting* subfamilies.

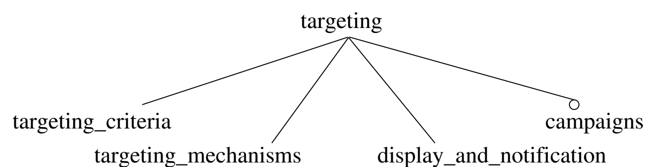

Figure 5.12: Targeting Overview

Affiliates

The *affiliates* subfamily deals with business partners who drive customers to the website. Moreover, it deals with commissioning those partners. Figure 5.13 shows the subfamily and its two features.

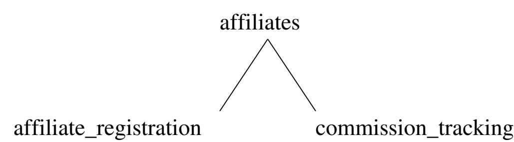

Figure 5.13: Affiliates Overview

Inventory Tracking

The *inventory_tracking* subfamily allows the e-shop staff to track the inventory stocks. Figure 5.14 shows the *inventory_tracking* subfamily.

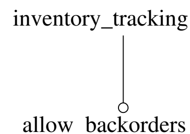

Figure 5.14: Inventory Tracking Overview

Procurement

The *procurement* subfamily deals with buying products from other business partners. Figure 5.15 shows the subfamily and its features.

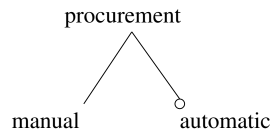

Figure 5.15: Procurement Overview

Reporting and Analysis

The *reporting_and_analysis* subfamily allows the staff of the e-shop to generate reports and analyze the data collected. These reports helps in developing a better business strategies and assessing the company's performance. Figure 5.16 shows the subfamily and its features.

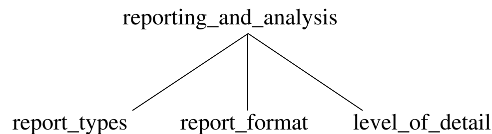

Figure 5.16: Reporting and Analysis Overview

External Systems Integration

The *external_system_integration* subfamily allows the company to integrate external systems to the e-shop system. The external system can be one or more of any of the following systems: *fulfillment_system*, *inventory_management_system*, *procurement_system*, or *external_ distributor_system*. Figure 5.17 shows the *external_systems_integration* subfamily.

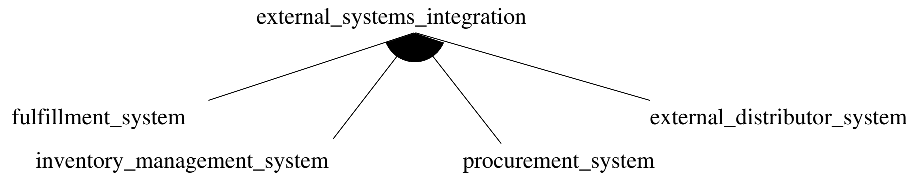

Figure 5.17: External Systems Integration Overview

Administration

The *administration* subfamily is responsible for operating and managing the e-shop system. Moreover, it has the configuration options for the system. Figure 5.18 shows an overview of the *administration* subfamily.

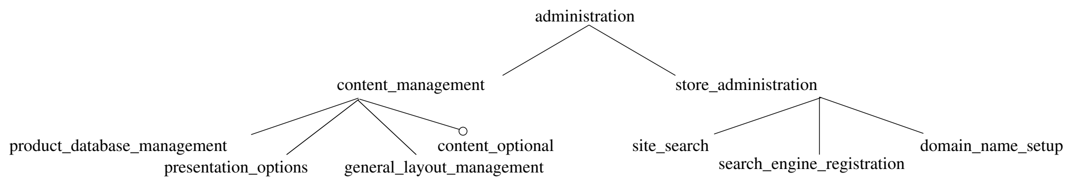

Figure 5.18: Administration Overview

5.2 E-shop Specification

In this section, I present the specification of the e-shop system written in the proposed language. To present the specification in an elegant way and to take advantage of the modularity and abstraction of the language, I split the specification into three main files. The first file contains the specification for the *store_front*. The second file contains the specification for the *business_management* subfamily. The third file constraints the specification for *e-shop* system. The reminder of the system specifications are presented in Appendix C.

Listing 5.1 is the specification of the subfamily *store_front* presented in Section 5.1.1. The specification includes the specifications of its subfamilies, defines the *store_front* subfamily, and specifies constraints to the family.

```
1  include home_page.spec
2  include registration.spec
3  include catalog.spec
4  include wish_list.spec
5  include buy_paths.spec
6  include customer_service.spec
7  include user_behaviour_tracking.spec
8
9  //----- store front
10 store_front = all_of(opt(home_page), opt(registration), catalog, opt
       (wish_list), buy_paths, opt(customer_service), opt(
       user_behaviour_tracking))
11
12 in store_front, wish_list require wish_list_save_after_session
13 in store_front, electronic_goods require product_size
14 in store_front, physical_goods require product_size
15 in store_front, user_behaviour_tracking_information require
       user_behaviour_tracking
16 in store_front, quick_checkout require quick_checkout_profile
17 in store_front, registered_checkout require register_to_buy
18 in store_front, registered_checkout require registration_enforcement
19 in store_front, physical_goods require product_weight
20 in store_front, category_page require categories
21 in store_front, permissions require registration
```

```
22 in store_front, email_wish_list require registration
```

Listing 5.1: The *store_front* Specification

Listing 5.2 is the specification of *business_management* subfamily which presented in Section 5.1.2. The specification file includes the specifications of its subfamilies and the definition of *business_management* in terms of its subfamilies.

```
1  include order_management.spec
2  include targeting
3  include affiliates.spec
4  include inventory_tracking.spec
5  include procurement.spec
6  include reporting_and_analysis.spec
7  include external_systems_integration.spec
8  include administration.spec
9
10 //----- business management
11 business_management = all_of(order_management, opt(targeting), opt(
      affiliates), opt(inventory_tracking), opt(procurement), opt(
      reporting_and_analysis), opt(external_systems_integration),
      administration)
```

Listing 5.2: The *business_management* Specification

Listing 5.3 is the file that contains the specification of the *e-shop* system presented in Section 5.1. The file glues the two previous specifications into *e-shop* family. Moreover, it has the constraints of the system.

```
1  include store_front.spec
```

```
2  include business_management.spec
3
4  //----- eShop
5  eShop = all_of(store_front , business_management)
6
7  in eShop, electronic_goods require electronic_goods_fulfillment
8  in eShop, customer_preferences require prefrences
9  in eShop, services require services_fulfillment
10 in eShop, physical_goods require physical_goods_fulfillment
11 in eShop, wish_list_content require wish_list
12 in eShop, shipping_options require shipping
13 in eShop, previously_visited_pages require pages
14 in eShop, special_offers require discounts
15 in eShop, product_availability require inventory_tracking
```

Listing 5.3: The *e-shop* Specification

5.3 Assessment and Results

In this section, I assess three *eShop* subfamily specifications on the tool Jory. The three families are *customer_service*, *registration*, and *targeting*. They vary in the number of basic features and families. In the following, I present the specifications, compile them, perform queries on them, and show the results.

5.3.1 Customer Service

The *customer_service* subfamily is presented in Section 5.1.1. The specification is presented in Appendix C. The *customer_service* is a small family. It contains 8 basic features and 5 families.

First, I loaded the specification in Jory, I chose the *set* model, and compiled the specification by pressing the button *compile*. The message "Compilation Completed" indicates the finish of compilation. Since no errors are found in the specifications, no error message was shown. Figure 5.19 shows the tool after compiling the specification.

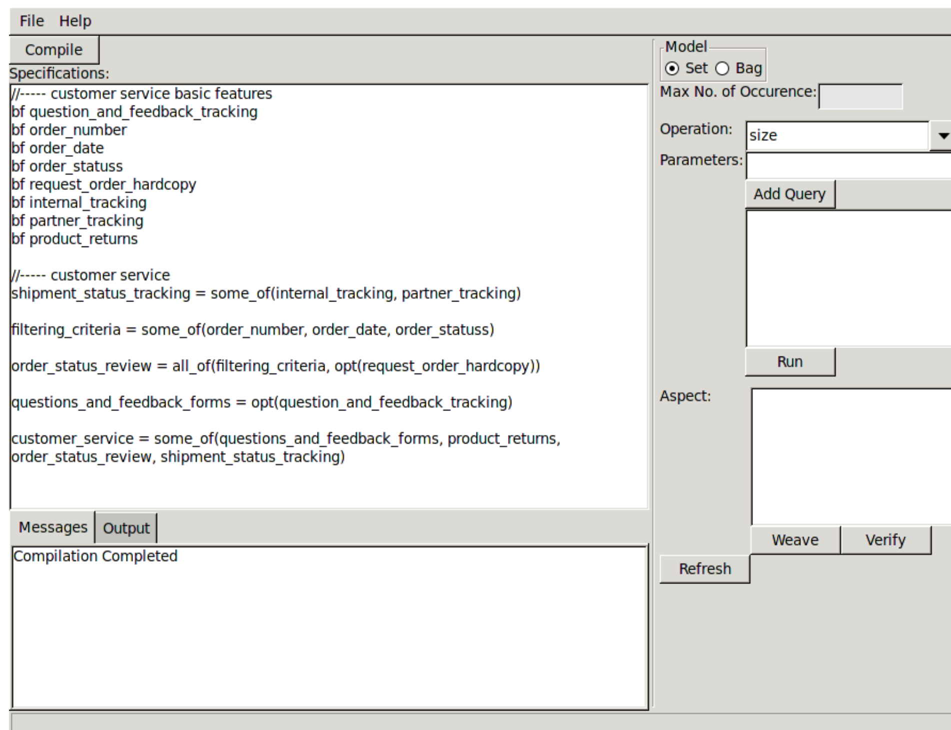

Figure 5.19: Customer Service on Jory After Compilation

The next step is to perform some queries on the compiled specification. I selected the

queries "*size*", "*listCommonality*", and "*listProducts*" on the family *customer_service*. The size of the family is 240, there is no commonality of features between the products in the family, and the list of products is shown on the interface as shown in Appendix D. Figure 5.20 shows the interface of the Jory after running the queries.

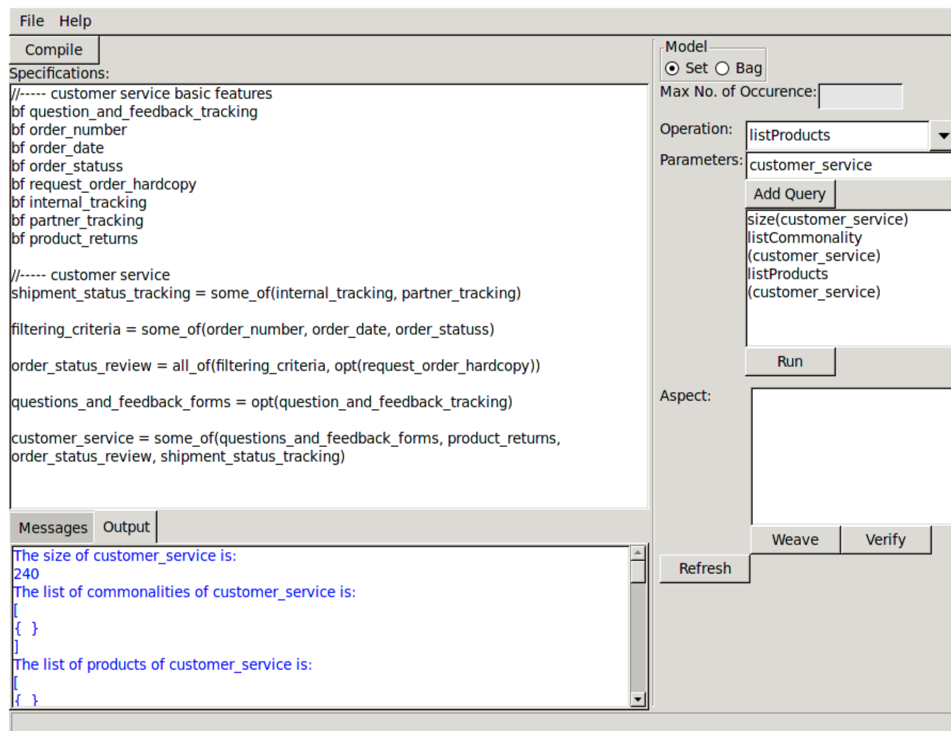

Figure 5.20: Customer Service on Jory After Executing Queries Part 1

I performed the queries "*refines*" and "*isSubfamily*" to check if the family *filtering_criteria* refines and subfamily of the family *order_status_review*. The result of the both queries is *yes* as shown in Figure 5.21.

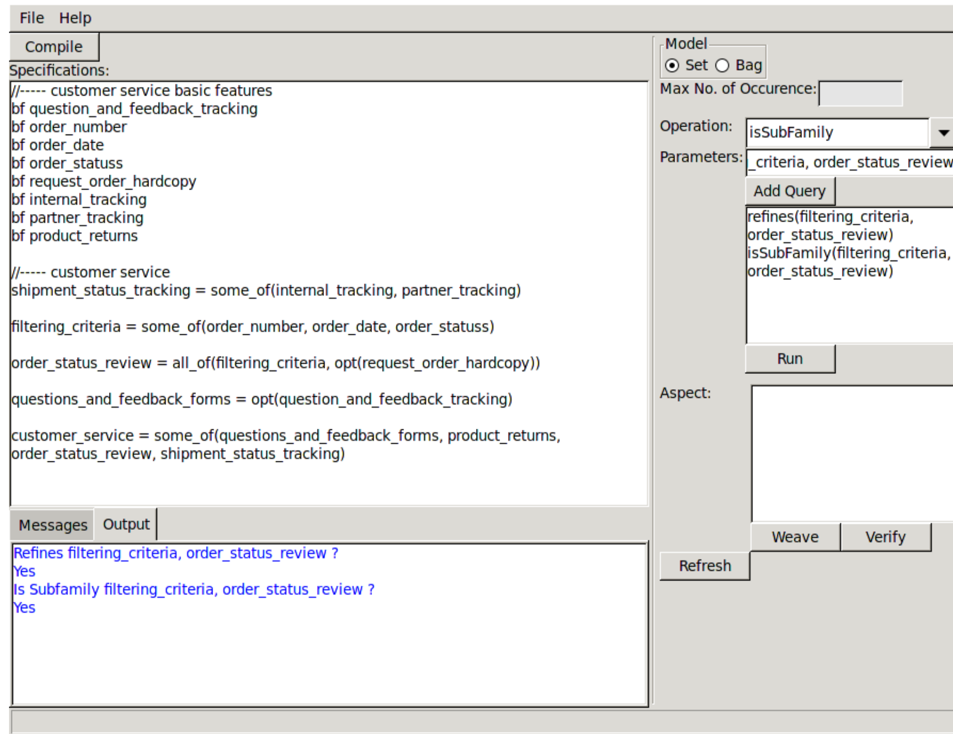

Figure 5.21: Customer Service on Jory After Executing Queries Part 2

5.3.2 Registration

The *registration* subfamily is presented in Section 5.1.1 and the specification is presented in Appendix C. The *registration* subfamily is larger than the *customer_service* family. It contains 22 basic features and 8 families.

I loaded the specification into Jory and compiled the specification on the *set* model. After compilation, I performed the following queries: "*size*" on the family *registration*, "*listProducts*" on the family *credit_card_information*, and "*isEqual*" to check the equality of *billing_address* and *shipping_address*. Figure 5.22 shows Jory interface after executing the queries on the *registration* specification. The "*size*" of *registration* is 344064. The family *credit_card_information* has the two products as shown below.

The families *billing_address* and *shipping_address* are not equal.

```
[
{ card_holder_name, card_number, expiry_date }
{ card_holder_name, card_number, expiry_date, security_information }
]
```

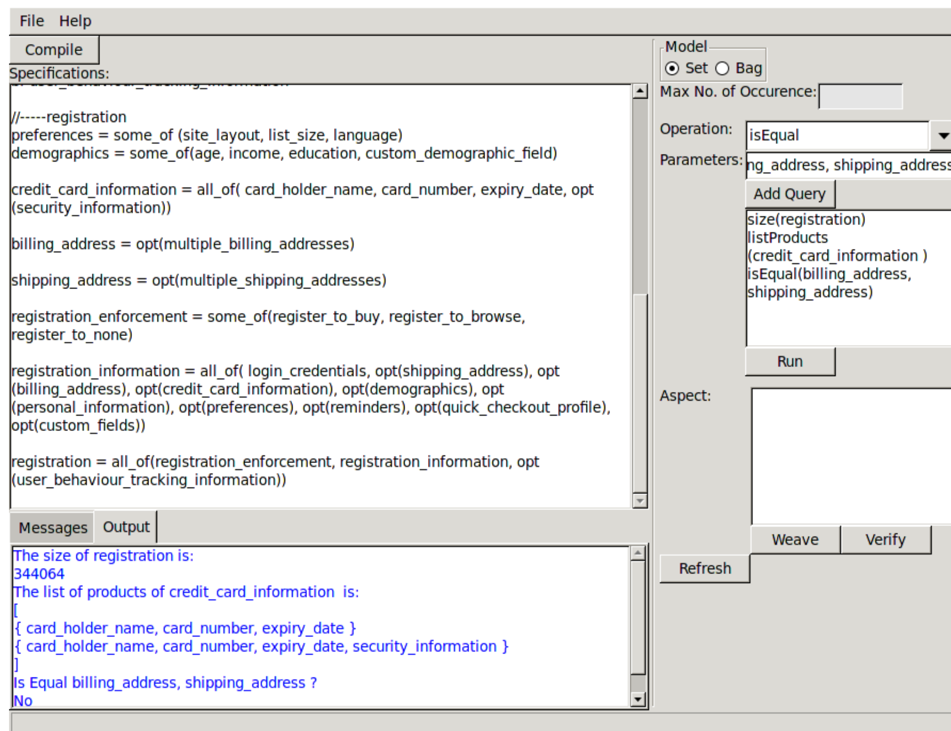

Figure 5.22: Registration on Jory After Executing

5.3.3 Targeting

I have introduced the family *targeting* in Section 5.1.2. The specification of the family is presented in Appendix C. It consists of 33 basic features and 13 families.

I loaded the specification in Jory and compiled it on the *set* model. Afterwards, I executed the query "*size*" on the family *targeting*. I also checked if *targeting_mechanisms* refines or is a subfamily of *advertisements*. The number of products in *targeting* is 1153650 products. Figure 5.23 shows the results after executing the queries on the *targeting* specification. The family *targeting_mechanisms* does not refine *advertisements* but it is a subfamily of *advertisements*.

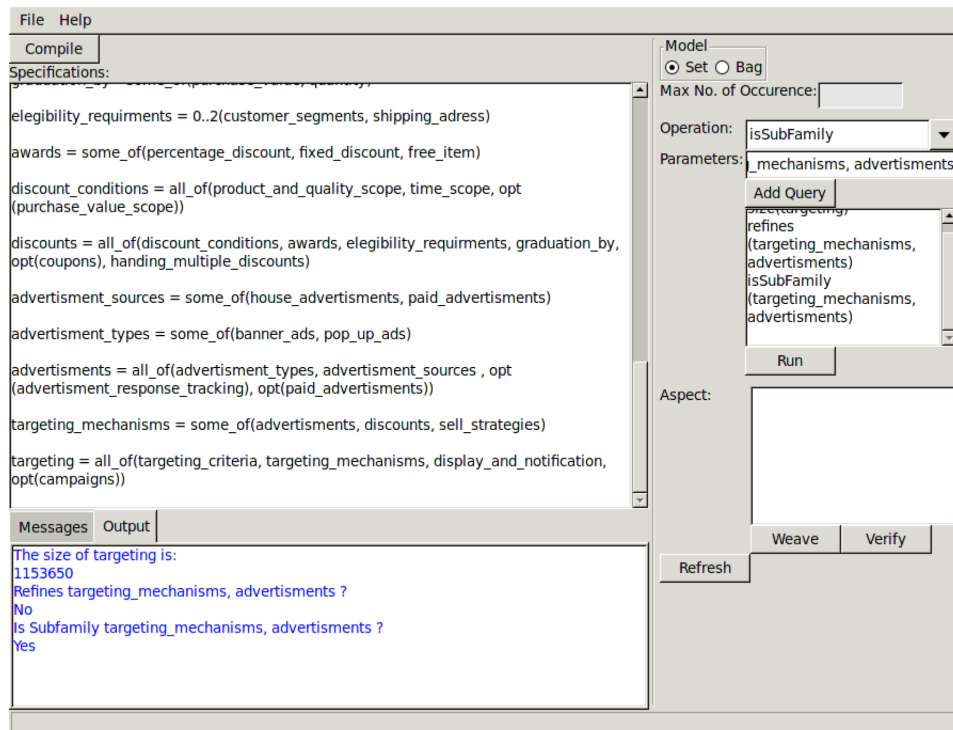

Figure 5.23: Targeting on Jory After Executing

5.4 Conclusion

In this chapter, I presented an industrial case study of an e-shop system. I showed the specification written in the proposed language. Moreover, I compiled the specification on the tool Jory and performed some tests and queries on the compiled specification. This case study illustrated that the the proposed language and Jory are applicable for large industrial examples.

Chapter 6

Conclusion

In this thesis, I proposed a feature modelling language and considered the design and implementation of its compiler. The language is intended to be readable, writable, modular, expressive, and have a deterministic interpretation. Moreover, the language has a compiler that checks the syntax of specifications, reports errors, and handles exceptions. The developed language is based on PFA and is implemented as a part of the tool Jory. This work fulfils the need for a feature modelling language that has an easy syntax and a strong mathematical foundation.

The language proposed in this thesis is precise since it has the advantage of the solid mathematical background of PFA. PFA is an algebraic feature modelling technique that is based on the mathematical structure of an idempotent and commutative semiring. It allows to capture and analyze features of product families. Moreover, the tool Jory, which is is a feature modelling tool built based on PFA and BDDs, makes mathematical manipulation fast because it uses BDDs to implement PFA models (i.e., set and bag). BDDs use memory with efficiency.

The proposed language solves the limitation of the previous techniques by having an easy syntax, rich constructs, and a solid background (i.e., by translating specifications to PFA).

6.1 Contributions

To design the language I had performed the following tasks:

1. Provided the design of the language (i.e., syntax and semantics).
2. Built the compiler and connected it to the tool Jory.
3. Assessed and tested the system by performing a case study.

The syntax of the language is simple and natural. This makes reading, writing, and modifying specifications an easy task. Moreover, the inclusion construct allows for reuse of already written specifications and provides a high level of abstraction.

The compiler of the language accepts specifications written in the proposed language, checks the syntax of the written specifications, reports errors, and generates the equivalent PFA specifications. This makes the written specifications more reliable and robust (i.e., error free). Building the compiler as a part of the tool Jory makes the system a black box that performs the tasks and generates results while hiding the details from the user.

The assessment of the language and tool presented in this thesis shows that the proposed language and the tool Jory are applicable to real-world applications, and testing

the system and shows that it generates the expected results.

6.2 Future Work

The proposed language and the tool allow for extension and improvement. The language's design allows for new constructs to be added to the language. It also allows for editing the current keywords and structure. Moreover, the layered architecture of Jory permits the addition of new layers and the removal and editing of existing layers.

One of the areas that can be improved in the tool is implementing the constraints in the *Term Evaluation Layer* and make the needed changes to the affected layers.

Another suggested improvement is the implementation of the *Translation Layer* in the design of Jory [Alt10]. The *Translation Layer* allows the translation between graphical feature modelling techniques and PFA language and vice versa.

Another possible future work is to make the tool Jory available online. This would make the tool accessible for interested users without the need to install the tool locally. Moreover, it would create a repository of realistic case studies created by users from different domains and specialities.

Appendix A

Feature Modelling Techniques Constructs

Table A.1, shows the different feature modelling constructs in several techniques. The constructs are mandatory feature, optional feature, AND relation, XOR relation, OR relation, requires constraint, and excludes constraint. The feature modelling techniques presented are FODA [KCH+90], FORM [KKL+98], FOPLE [KL02], FeatuRSEB [GFA98], GP [CE00], Van Gurp [vGBS01], Riebisch [RBSP02], PLUSS [EBB05], and the Proposed Language.

Expression	FODA	FORM	FOPLE	FeatuRSEB	GP	Van Gurp	Riebisch	PLUSS	Proposed Language
Feature	a	a	a	a	a	a	a	a	a
Mandatory Feature	a	a	a	a	a	a	a	a	a
Optional Feature	a	a	a	a	a	a	a	a	**opt**(a)
AND Relation	f a b c	f a b c	f a b c	f a b c	f a b c	f a b c	f a b c	f a b c	f =**all_of**(a, b, c)
XOR Relation	f a b c	f a b c	f a b c	f a b c	f a b c	f a b c	f a b c	f S S S a b c	f =**one_of**(a, b, c)
OR Relation	N/A	N/A	N/A	f a b c	f a b c	f a b c	f a b c	f M M M a b c	f =**some_of**(a, b, c)
Requires Constraint	textual	textual	textual	a requires b	textual	a Requires b	a <<requires>> b	a <<requires>> b	**in** f, a **require** b
Excludes Constraint	textual	textual	textual	a mutex b	textual	a Excludes b	a <<excludes>> b	a <<excludes>> b	**in** f, a **exclude** b

Table A.1: Constructs of Feature Modelling Techniques

Appendix B

Sample Code: Family Generator

```
1   -- | This module reads the intermediate code for families and
        generates the equivalent PFA code
2  module FamilyGenerator where
3  import System.IO
4  import System.Environment
5  import Data.List
6  import Data.Char
7  import FileHandler
8  import Line
9
10 -- | This function reads a family file and generates code to the
       output file. It also, generates a file containing family names
       for later use
11 processfamily fm spec = do fileEnd <- hIsEOF fm
12             if (fileEnd) then return ()
13              else do line <- hGetLine fm
14                       setFile (spec) (processfmLine line)
15                setFile "../working_dir/family.txt" (getfn line)
```

```
16                  processfamily fm spec
17
18
19
20  -- | This function reads a family line, checks it, then returns the
        code
21  processfmLine :: String -> String
22  processfmLine xs
23       | ('(' /= head(xs)) = xs
24       | otherwise = processfun xs
25
26  -- | This function processes a line list
27  processfun :: String -> String
28  processfun xs = fun (getline(rmParens xs))
29
30
31  -- | This function takes a line list and returns its PFA code
32  fun :: [String] -> String
33  fun (x:xs)
34       | (x == "=") = equal (x:xs)
35       | (x == "+") = plusDot (x:xs)
36       | (x == ".") =  plusDot (x:xs)
37       | (x == "all_of") = allOf (xs)
38       | (x == "one_of") = oneOf (xs)
39       | (x == "some_of") = someOf (subsequences xs)
40       | (isDigit (head x)) = cardinality (x:xs)
41       | (x == "^")  = power xs
42       | ( x == "opt") = optional xs
```

```
43       | ( '(' == (head x)) && (')' == (last x)) = "( " ++ (
            processfmLine x) ++ " )"
44       | otherwise = x
45
46 -- | This function processes the = function
47 equal (f:p1:p2) = p1 ++ " " ++ f ++ " " ++ (processfmLine (concat
      p2)) ++ " % " ++ p1
48
49 -- | This function processes the + and . functions
50 plusDot (f: p1: p2) = (processfmLine p1) ++ " " ++ f ++ " " ++ (
      processfmLine (concat p2))
51
52 -- | This function processes the all_of function
53 allOf [] = []
54 allOf (x:xs)
55         |(xs == []) = (processfmLine x)
56         | otherwise = (processfmLine x) ++ " . " ++ (allOf xs)
57
58 -- | This function processes the one_of function
59 oneOf [] = []
60 oneOf (x:xs)
61         |(xs == []) = (processfmLine x)
62         | otherwise = (processfmLine x) ++ " + " ++ (oneOf xs)
63
64 -- | This function processes the some_of function
65 someOf [] = []
66 someOf (x:xs)
67         | (x == []) = someOf xs
68         | (xs == []) = allOf x
```

```
69         | otherwise = (allOf x) ++ " + " ++ (someOf xs)
70
71  -- | This function processes the cardinality
72  cardinality (x: xs)
73       | (isInfixOf ".." x) = someOf(function (digitoInt  x') (
            digitoInt (dropWhile (=='.') x'')) (subsequences xs))
74       | otherwise = someOf(function (digitoInt x) (digitoInt x) (
            subsequences xs))
75         where (x',x'') = span (/='.') x
76  -- |
77  function n1 n2 [] = []
78  function n1 n2 (x:xs)
79       | ((length x) >= n1) && ((length x) <= n2) = x: function n1 n2
            xs
80       | otherwise = function n1 n2 xs
81  -- |
82  power (x:xs:u)
83     | (isInfixOf ".." xs) = repeats (digitoInt xs') (digitoInt (
          dropWhile (=='.') xs'')) x
84     | otherwise =  repeatt (digitoInt  xs) x
85       where (xs',xs'') = span (/='.') xs
86
87  -- |
88  repeats :: Int -> Int -> String -> String
89  repeats n m x
90     | (n == m) = (repeatt n x)
91     | otherwise = (repeatt n x) ++ " + " ++ (repeats (n+1) m x)
92  -- |
93  repeatt n x
```

```
94      | (n == 1) = (processfmLine x)
95      | otherwise = (processfmLine x) ++ " . " ++ (repeatt (n-1) x)
96
97  -- | This function processes the opt function
98  optional x = "( "++ processfmLine(concat x) ++" + 1)"
99
100 getfn xs = fn(getline(rmParens xs))
101
102 fn ( f : x :xs) = x
```

Appendix C

E-shop Specifications

C.1 home_page.spec

```
1  //----- home page basic features
2  bf static_content
3  bf welcome_message
4  bf special_offers
5  bf time_dependent
6  bf personalized
7
8  //----- home page
9  content_type = some_of( welcome_message, special_offers)
10
11 variation_source = some_of( time_dependent, personalized)
12
13 dynamic_content = all_of(content_type, variation_source)
14
15 home_page = some_of( static_content, dynamic_content)
```

C.2 registration.spec

```
1   //----- registration basic features
2   bf register_to_buy
3   bf register_to_browse
4   bf register_to_none
5   bf multiple_shipping_addresses
6   bf multiple_billing_addresses
7   bf card_holder_name
8   bf card_number
9   bf expiry_date
10  bf security_information
11  bf age
12  bf income
13  bf education
14  bf custom_demographic_field
15  bf site_layout
16  bf list_size
17  bf language
18  bf login_credentials
19  bf personal_information
20  bf reminders
21  bf quick_checkout_profile
22  bf cusom_fields
23  bf user_behaviour_tracking_information
24
25  //-----registration
26  preferences = some_of (site_layout , list_size , language)
27  demographics = some_of(age , income , education ,
        custom_demographic_field)
```

```
28
29 credit_card_information = all_of( card_holder_name, card_number,
       expiry_date, opt(security_information))
30
31 billing_address = opt(multiple_billing_addresses)
32
33 shipping_address = opt(multiple_shipping_addresses)
34
35 registration_enforcement = some_of(register_to_buy,
       register_to_browse, register_to_none)
36
37 registration_information = all_of( login_credentials, opt(
       shipping_address), opt(billing_address), opt(
       credit_card_information), opt(demographics), opt(
       personal_information), opt(preferences), opt(reminders), opt(
       quick_checkout_profile), opt(cusom_fields))
38
39 registration = all_of(registration_enforcement,
       registration_information, opt(user_behaviour_tracking_information
       ))
```

C.3 registration.spec

```
1 //----- catalog basic features
2 bf electronic_goods
3 bf physical_goods
4 bf services
5 bf thumbnail
6 bf image_2d
```

```
7  bf image_3d
8  bf image_360
9  bf different_perspective
10 bf gallery
11 bf video
12 bf sound
13 bf documents
14 bf complex_product_configuration
15 bf basic_information
16 bf detailed_information
17 bf warranty_information
18 bf customer_reviews
19 bf product_size
20 bf product_weight
21 bf product_availability
22 bf product_custom_fields
23 bf multi_level
24 bf multiple_classification
25 bf basic_search
26 bf advanced_search
27 bf price
28 bf quality
29 bf price_quality
30 bf manufacturer_name
31 bf custom_filter
32 bf product_page
33 bf category_page
34 bf seasonal_product_views
35 bf personalized_views
```

```
36 bf multiple_catalogs
37
38 //----- catalog
39 custom_views = some_of(seasonal_product_views, personalized_views)
40
41 sorting_filters = some_of(price, quality, manufacturer_name,
       custom_filter)
42
43 index_page = opt(sorting_filters)
44
45 browsing = all_of( product_page, opt(category_page), opt(index_page)
       )
46
47 searching = some_of(basic_search, advanced_search)
48
49 catalog_categories = some_of(multi_level, multiple_classification)
50
51 categories = opt(catalog_categories)
52
53 product_variants = opt(complex_product_configuration)
54
55 image = some_of(thumbnail, image_2d, image_3d, image_360,
       different_perspective, gallery)
56
57 media_files = some_of(image, video, sound)
58
59 associated_assets = some_of( documents, media_files)
60
61 product_type = some_of(electronic_goods, physical_goods, services)
```

```
62
63 product_information= all_of(product_type, basic_information, opt(
       detailed_information), opt(warranty_information), opt(
       customer_reviews), opt(associated_assets), opt(product_variants),
        opt(product_size), opt(product_weight), opt(product_availability
       ), opt(product_custom_fields))
64
65 catalog = all_of(product_information, opt(categories), opt(
       multiple_catalogs), opt(searching), opt(browsing), opt(
       custom_views))
```

C.4 wish_list.spec

```
 1   //----- wish list basic features
 2 bf public_access
 3 bf restricted_access
 4 bf private_access
 5 bf wish_list_save_after_session
 6 bf email_wish_list
 7 bf multiple_wish_list
 8
 9 //----- wish list
10 permissions = some_of(public_access, restricted_access,
       private_access)
11
12 wish_list = some_of(wish_list_save_after_session, email_wish_list,
       multiple_wish_list, permissions)
```

C.5 buy_paths.spec

```
1      //----- buy path basic features
2   bf inventory_management_policy
3   bf cart_content_page
4   bf cart_page_summary
5   bf cart_save_after_session
6   bf enable_profile_update_on_checkout
7   bf guest_checkout
8   bf quality_of_service_selection
9   bf carrier_selection
10  bf gift_options
11  bf multiple_shipments
12  bf shipping_cost_calculation
13  bf country
14  bf region
15  bf city
16  bf shipping
17  bf billing
18  bf tax_codes
19  bf fixed_rate_taxation
20  bf surcharge
21  bf percentage
22  bf certiTAX
23  bf cyberSource
24  bf custom_tax_gateway
25  bf cod
26  bf credit_card
27  bf debit_card
28  bf electronic_cheque
29  bf fax_mail_order
```

```
30 bf purchase_order
31 bf gift_certificate
32 bf phone_order
33 bf custom_payment_type
34 bf fraud_detection
35 bf payment_types
36 bf authorizeNet
37 bf linkpoint
38 bf paradata
39 bf skipJack
40 bf version_payflow_pro
41 bf payment_gateways
42 bf electronic_page
43 bf email
44 bf phone
45 bf mail
46 bf digital_dialing
47 bf rotary_dialing
48
49 //----- buy path
50 phone_ordering = all_of(digital_dialing, opt(rotary_dialing))
51
52 order_confirmation = some_of(electronic_page, email, phone, mail)
53
54 payment_options = all_of(payment_types, opt(fraud_detection), opt(
       payment_gateways))
55
56 tax_gateways = some_of(certiTAX, cyberSource, custom_tax_gateway)
57
```

```
58 amount_specification = some_of(surcharge, percentage)
59
60 address = all_of(shipping, opt(billing))
61
62 resolution = some_of(country, region, city)
63
64 ruled_based_taxation = all_of(tax_codes, address, resolution)
65
66 type = some_of(fixed_rate_taxation, ruled_based_taxation)
67
68 custom_taxation = all_of(type, amount_specification)
69
70 taxation_options = some_of(custom_taxation, tax_gateways)
71
72 shipping_options = all_of(opt(quality_of_service_selection), opt(
       carrier_selection), opt(gift_options), opt(multiple_shipments),
       shipping_cost_calculation)
73
74
75 quick_checkout = opt(enable_profile_update_on_checkout)
76
77 registered_checkout = opt(quick_checkout)
78
79 checkout_type = some_of(registered_checkout, guest_checkout)
80
81 checkout = all_of(checkout_type, opt(shipping_options),
       taxation_options, payment_options)
82
```

```
83 shopping_cart = all_of(inventory_management_policy,
       cart_content_page, opt(cart_page_summary), opt(
       cart_save_after_session))
84
85 buy_paths = all_of(shopping_cart, checkout, order_confirmation,
       phone_ordering)
```

C.6 customer_service.spec

```
 1     //----- customer service basic features
 2 bf question_and_feedback_tracking
 3 bf order_number
 4 bf order_date
 5 bf order_status
 6 bf request_order_hardcopy
 7 bf internal_tracking
 8 bf partner_tracking
 9 bf product_returns
10
11 //----- customer service
12 shipment_status_tracking = some_of(internal_tracking,
       partner_tracking)
13
14 filtering_criteria = some_of(order_number, order_date, order_status)
15
16 order_status_review = all_of(filtering_criteria, opt(
       request_order_hardcopy))
17
18 questions_and_feedback_forms = opt(question_and_feedback_tracking)
```

```
19
20 customer_service = some_of(questions_and_feedback_forms ,
     product_returns , order_status_review , shipment_status_tracking)
```

C.7 user_behaviour_tracking.spec

```
1 //----- user behaviour tracking basic features
2 bf locally_visited_pages
3 bf external_referring_pages
4 bf previous_purchases
5
6 //----- user behaviour tracking
7 pages = one_of(locally_visited_pages , external_referring_pages) //to
      be used for eShop constraint
8
9 user_behaviour_tracking = some_of(locally_visited_pages ,
     external_referring_pages , previous_purchases)
```

C.8 order_management.spec

```
1 //-----order management basic features
2 bf quality_purchased
3 bf order_total
4 bf order_weight
5 bf product_classification
6 bf flat_rate
7 bf fedEx
8 bf ups
9 bf usps
10 bf canada_post
```

```
11 bf custom_shipping_gateway
12 bf shipping_gateway
13 bf warehouse_management
14 bf file_repository
15 bf license_management
16 bf appointment_scheduling
17 bf resource_planning
18
19 //----- order management
20 services_fulfillment = 0..2(appointment_scheduling,
     resource_planning)
21
22 electronic_goods_fulfillment = all_of(file_repository,
     license_management)
23
24 rate_factors = some_of(quality_purchased, order_weight,
     product_classification)
25
26 pricing = all_of(flat_rate, opt(rate_factors))
27
28 order_shipping = some_of(pricing, shipping_gateway)
29
30 physical_goods_fulfillment = all_of(warehouse_management,
     order_shipping)
31
32 order_management = some_of(physical_goods_fulfillment,
     electronic_goods_fulfillment, services_fulfillment)
```

C.9 targeting.spec

```
1  //----- targeting basic features
2  bf customer_preferences
3  bf shopping_cart_content
4  bf wish_list_content
5  bf previously_visited_pages
6  bf date_and_time
7  bf custom_target_criteria
8  bf targeting_criteria
9  bf banner_ads
10 bf pop_up_ads
11 bf house_advertisements
12 bf paid_advertisements
13 bf advertisement _response_tracking
14 bf context_sensitive_ads
15 bf product_and_quality_scope
16 bf time_scope
17 bf purchase_value_scope
18 bf percentage_discount
19 bf fixed_discount
20 bf free_item
21 bf customer_segments
22 bf shipping_address
23 bf purchase_value
24 bf quantity
25 bf coupons
26 bf handing_multiple_discounts
27 bf product_kitting
28 bf up_selling
```

```
29 bf cross_selling
30 bf personalized_email
31 bf response_tracking
32 bf assignment_to_page_types_for_ display
33 bf product_flagging
34 bf campaigns
35
36 //----- targeting
37 emails = 0..2(personalized_email , response_tracking)
38
39 display_and_notification = some_of(
      assignment_to_page_types_for_display , product_flagging , emails)
40
41 sell_strategies = some_of(product_kitting , up_selling , cross_selling
      )
42
43 graduation_by= some_of(purchase_value , quantity)
44
45 eligibility_requirements = 0..2(customer_segments , shipping_address)
46
47 awards = some_of(percentage_discount , fixed_discount , free_item)
48
49 discount_conditions = all_of(product_and_quality_scope , time_scope ,
      opt(purchase_value_scope))
50
51 discounts = all_of(discount_conditions , awards ,
      eligibility_requirements , graduation_by , opt(coupons) ,
      handing_multiple_discounts)
52
```

```
53 advertisement_sources = some_of(house_advertisements,
       paid_advertisements)
54
55 advertisement_types = some_of(banner_ads, pop_up_ads)
56
57 advertisements = all_of(advertisement_types, advertisement_sources ,
        opt(advertisement_response_tracking), opt(paid_advertisements))
58
59 targeting_mechanisms = some_of(advertisements, discounts,
       sell_strategies)
60
61 targeting = all_of(targeting_criteria, targeting_mechanisms,
       display_and_notification, opt(campaigns))
```

C.10 affiliates.spec

```
1 //----- affiliates basic features
2 bf affiliates_registration
3 bf commission_tracking
4
5 //----- affiliates
6 affiliates = all_of(affiliates_registration, commission_tracking)
```

C.11 inventory_tracking.spec

```
1 //----- inventory tracking basic features
2 bf allow_backorders
3
4 //----- inventory tracking
5 inventory_tracking = opt(allow_backorders)
```

C.12 procurement.spec

```
1 //----- procurement basic features
2 bf manual
3 bf automatic
4
5 //----- procurement
6 procurement = all_of(manual, opt(automatic))
```

C.13 reporting_and_analysis.spec

```
1 //----- reporting and analysis basic features
2 bf report_types
3 bf report_format
4 bf level_of_detail
5
6 //----- reporting and analysis
7 reporting_and_analysis = all_of(report_types, report_format,
      level_of_detail)
```

C.14 external_systems_integration.spec

```
1 //----- external systems integration basic features
2 bf fulfillment_system
3 bf inventory_management_system
4 bf procurement_system
5 bf external_distributor_system
6
7 //----- external systems integration
```

```
8  external_systems_integration = some_of(fulfillment_system,
       inventroy_management_system, procurement_system,
       external_distributor_system)
```

C.15 administration.spec

```
1  //----- administration basic features
2  bf product_database_management
3  bf presentation_options
4  bf general_layout_management
5  bf content_approval
6  bf site_search
7  bf search_engine_registration
8  bf domain_name_setup
9
10 //----- administration
11 store_administration = all_of(site_search,
       search_engine_registration, domain_name_setup)
12
13 content_management = all_of(product_database_management,
       presentation_options, general_layout_management, opt(
       content_approval))
14
15 administration = all_of(content_management, store_administration)
```

Appendix D

Customer Service Result

```
The list of products of customer_service is:
[
{ }
{ product_returns }
{ partner_tracking }
{ partner_tracking, product_returns }
{ internal_tracking }
{ internal_tracking, product_returns }
{ internal_tracking, partner_tracking }
{ internal_tracking, partner_tracking, product_returns }
{ order_status }
{ order_status, product_returns }
{ order_status, partner_tracking }
{ order_status, partner_tracking, product_returns }
{ order_status, internal_tracking }
{ order_status, internal_tracking, product_returns }
{ order_status, internal_tracking, partner_tracking }
{ order_status, internal_tracking, partner_tracking, product_returns }
```

```
{ order_status , request_order_hardcopy }
{ order_status , request_order_hardcopy , product_returns }
{ order_status , request_order_hardcopy , partner_tracking }
{ order_status , request_order_hardcopy , partner_tracking ,
   product_returns }
{ order_status , request_order_hardcopy , internal_tracking }
{ order_status , request_order_hardcopy , internal_tracking ,
   product_returns }
{ order_status , request_order_hardcopy , internal_tracking ,
   partner_tracking }
{ order_status , request_order_hardcopy , internal_tracking ,
   partner_tracking , product_returns }
{ order_date }
{ order_date , product_returns }
{ order_date , partner_tracking }
{ order_date , partner_tracking , product_returns }
{ order_date , internal_tracking }
{ order_date , internal_tracking , product_returns }
{ order_date , internal_tracking , partner_tracking }
{ order_date , internal_tracking , partner_tracking , product_returns }
{ order_date , request_order_hardcopy }
{ order_date , request_order_hardcopy , product_returns }
{ order_date , request_order_hardcopy , partner_tracking }
{ order_date , request_order_hardcopy , partner_tracking , product_returns
   }
{ order_date , request_order_hardcopy , internal_tracking }
{ order_date , request_order_hardcopy , internal_tracking , product_returns
    }
```

```
{ order_date , request_order_hardcopy , internal_tracking ,
   partner_tracking }
{ order_date , request_order_hardcopy , internal_tracking ,
   partner_tracking , product_returns }
{ order_date , order_status }
{ order_date , order_status , product_returns }
{ order_date , order_status , partner_tracking }
{ order_date , order_status , partner_tracking , product_returns }
{ order_date , order_status , internal_tracking }
{ order_date , order_status , internal_tracking , product_returns }
{ order_date , order_status , internal_tracking , partner_tracking }
{ order_date , order_status , internal_tracking , partner_tracking ,
   product_returns }
{ order_date , order_status , request_order_hardcopy }
{ order_date , order_status , request_order_hardcopy , product_returns }
{ order_date , order_status , request_order_hardcopy , partner_tracking }
{ order_date , order_status , request_order_hardcopy , partner_tracking ,
   product_returns }
{ order_date , order_status , request_order_hardcopy , internal_tracking }
{ order_date , order_status , request_order_hardcopy , internal_tracking ,
   product_returns }
{ order_date , order_status , request_order_hardcopy , internal_tracking ,
   partner_tracking }
{ order_date , order_status , request_order_hardcopy , internal_tracking ,
   partner_tracking , product_returns }
{ order_number }
{ order_number , product_returns }
{ order_number , partner_tracking }
{ order_number , partner_tracking , product_returns }
```

```
{ order_number, internal_tracking }
{ order_number, internal_tracking, product_returns }
{ order_number, internal_tracking, partner_tracking }
{ order_number, internal_tracking, partner_tracking, product_returns }
{ order_number, request_order_hardcopy }
{ order_number, request_order_hardcopy, product_returns }
{ order_number, request_order_hardcopy, partner_tracking }
{ order_number, request_order_hardcopy, partner_tracking,
  product_returns }
{ order_number, request_order_hardcopy, internal_tracking }
{ order_number, request_order_hardcopy, internal_tracking,
  product_returns }
{ order_number, request_order_hardcopy, internal_tracking,
  partner_tracking }
{ order_number, request_order_hardcopy, internal_tracking,
  partner_tracking, product_returns }
{ order_number, order_status }
{ order_number, order_status, product_returns }
{ order_number, order_status, partner_tracking }
{ order_number, order_status, partner_tracking, product_returns }
{ order_number, order_status, internal_tracking }
{ order_number, order_status, internal_tracking, product_returns }
{ order_number, order_status, internal_tracking, partner_tracking }
{ order_number, order_status, internal_tracking, partner_tracking,
  product_returns }
{ order_number, order_status, request_order_hardcopy }
{ order_number, order_status, request_order_hardcopy, product_returns }
{ order_number, order_status, request_order_hardcopy, partner_tracking }
```

```
{ order_number, order_status, request_order_hardcopy, partner_tracking,
    product_returns }
{ order_number, order_status, request_order_hardcopy, internal_tracking
    }
{ order_number, order_status, request_order_hardcopy, internal_tracking,
     product_returns }
{ order_number, order_status, request_order_hardcopy, internal_tracking,
     partner_tracking }
{ order_number, order_status, request_order_hardcopy, internal_tracking,
     partner_tracking, product_returns }
{ order_number, order_date }
{ order_number, order_date, product_returns }
{ order_number, order_date, partner_tracking }
{ order_number, order_date, partner_tracking, product_returns }
{ order_number, order_date, internal_tracking }
{ order_number, order_date, internal_tracking, product_returns }
{ order_number, order_date, internal_tracking, partner_tracking }
{ order_number, order_date, internal_tracking, partner_tracking,
    product_returns }
{ order_number, order_date, request_order_hardcopy }
{ order_number, order_date, request_order_hardcopy, product_returns }
{ order_number, order_date, request_order_hardcopy, partner_tracking }
{ order_number, order_date, request_order_hardcopy, partner_tracking,
    product_returns }
{ order_number, order_date, request_order_hardcopy, internal_tracking }
{ order_number, order_date, request_order_hardcopy, internal_tracking,
    product_returns }
{ order_number, order_date, request_order_hardcopy, internal_tracking,
    partner_tracking }
```

```
{ order_number, order_date, request_order_hardcopy, internal_tracking,
    partner_tracking, product_returns }
{ order_number, order_date, order_status }
{ order_number, order_date, order_status, product_returns }
{ order_number, order_date, order_status, partner_tracking }
{ order_number, order_date, order_status, partner_tracking,
    product_returns }
{ order_number, order_date, order_status, internal_tracking }
{ order_number, order_date, order_status, internal_tracking,
    product_returns }
{ order_number, order_date, order_status, internal_tracking,
    partner_tracking }
{ order_number, order_date, order_status, internal_tracking,
    partner_tracking, product_returns }
{ order_number, order_date, order_status, request_order_hardcopy }
{ order_number, order_date, order_status, request_order_hardcopy,
    product_returns }
{ order_number, order_date, order_status, request_order_hardcopy,
    partner_tracking }
{ order_number, order_date, order_status, request_order_hardcopy,
    partner_tracking, product_returns }
{ order_number, order_date, order_status, request_order_hardcopy,
    internal_tracking }
{ order_number, order_date, order_status, request_order_hardcopy,
    internal_tracking, product_returns }
{ order_number, order_date, order_status, request_order_hardcopy,
    internal_tracking, partner_tracking }
{ order_number, order_date, order_status, request_order_hardcopy,
    internal_tracking, partner_tracking, product_returns }
```

```
{ question_and_feedback_tracking }
{ question_and_feedback_tracking , product_returns }
{ question_and_feedback_tracking , partner_tracking }
{ question_and_feedback_tracking , partner_tracking , product_returns }
{ question_and_feedback_tracking , internal_tracking }
{ question_and_feedback_tracking , internal_tracking , product_returns }
{ question_and_feedback_tracking , internal_tracking , partner_tracking }
{ question_and_feedback_tracking , internal_tracking , partner_tracking ,
   product_returns }
{ question_and_feedback_tracking , order_status }
{ question_and_feedback_tracking , order_status , product_returns }
{ question_and_feedback_tracking , order_status , partner_tracking }
{ question_and_feedback_tracking , order_status , partner_tracking ,
   product_returns }
{ question_and_feedback_tracking , order_status , internal_tracking }
{ question_and_feedback_tracking , order_status , internal_tracking ,
   product_returns }
{ question_and_feedback_tracking , order_status , internal_tracking ,
   partner_tracking }
{ question_and_feedback_tracking , order_status , internal_tracking ,
   partner_tracking , product_returns }
{ question_and_feedback_tracking , order_status , request_order_hardcopy }
{ question_and_feedback_tracking , order_status , request_order_hardcopy ,
   product_returns }
{ question_and_feedback_tracking , order_status , request_order_hardcopy ,
   partner_tracking }
{ question_and_feedback_tracking , order_status , request_order_hardcopy ,
   partner_tracking , product_returns }
```

```
{ question_and_feedback_tracking , order_status , request_order_hardcopy ,
   internal_tracking }
{ question_and_feedback_tracking , order_status , request_order_hardcopy ,
   internal_tracking , product_returns }
{ question_and_feedback_tracking , order_status , request_order_hardcopy ,
   internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_status , request_order_hardcopy ,
   internal_tracking , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_date }
{ question_and_feedback_tracking , order_date , product_returns }
{ question_and_feedback_tracking , order_date , partner_tracking }
{ question_and_feedback_tracking , order_date , partner_tracking ,
   product_returns }
{ question_and_feedback_tracking , order_date , internal_tracking }
{ question_and_feedback_tracking , order_date , internal_tracking ,
   product_returns }
{ question_and_feedback_tracking , order_date , internal_tracking ,
   partner_tracking }
{ question_and_feedback_tracking , order_date , internal_tracking ,
   partner_tracking , product_returns }
{ question_and_feedback_tracking , order_date , request_order_hardcopy }
{ question_and_feedback_tracking , order_date , request_order_hardcopy ,
   product_returns }
{ question_and_feedback_tracking , order_date , request_order_hardcopy ,
   partner_tracking }
{ question_and_feedback_tracking , order_date , request_order_hardcopy ,
   partner_tracking , product_returns }
{ question_and_feedback_tracking , order_date , request_order_hardcopy ,
   internal_tracking }
```

```
{ question_and_feedback_tracking , order_date , request_order_hardcopy ,
    internal_tracking , product_returns }
{ question_and_feedback_tracking , order_date , request_order_hardcopy ,
    internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_date , request_order_hardcopy ,
    internal_tracking , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_date , order_status }
{ question_and_feedback_tracking , order_date , order_status ,
    product_returns }
{ question_and_feedback_tracking , order_date , order_status ,
    partner_tracking }
{ question_and_feedback_tracking , order_date , order_status ,
    partner_tracking , product_returns }
{ question_and_feedback_tracking , order_date , order_status ,
    internal_tracking }
{ question_and_feedback_tracking , order_date , order_status ,
    internal_tracking , product_returns }
{ question_and_feedback_tracking , order_date , order_status ,
    internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_date , order_status ,
    internal_tracking , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy }
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy , product_returns }
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy , partner_tracking }
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy , partner_tracking , product_returns }
```

```
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy , internal_tracking }
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy , internal_tracking , product_returns }
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy , internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_date , order_status ,
    request_order_hardcopy , internal_tracking , partner_tracking ,
    product_returns }
{ question_and_feedback_tracking , order_number }
{ question_and_feedback_tracking , order_number , product_returns }
{ question_and_feedback_tracking , order_number , partner_tracking }
{ question_and_feedback_tracking , order_number , partner_tracking ,
    product_returns }
{ question_and_feedback_tracking , order_number , internal_tracking }
{ question_and_feedback_tracking , order_number , internal_tracking ,
    product_returns }
{ question_and_feedback_tracking , order_number , internal_tracking ,
    partner_tracking }
{ question_and_feedback_tracking , order_number , internal_tracking ,
    partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , request_order_hardcopy }
{ question_and_feedback_tracking , order_number , request_order_hardcopy ,
    product_returns }
{ question_and_feedback_tracking , order_number , request_order_hardcopy ,
    partner_tracking }
{ question_and_feedback_tracking , order_number , request_order_hardcopy ,
    partner_tracking , product_returns }
```

```
{ question_and_feedback_tracking , order_number , request_order_hardcopy ,
   internal_tracking }
{ question_and_feedback_tracking , order_number , request_order_hardcopy ,
   internal_tracking , product_returns }
{ question_and_feedback_tracking , order_number , request_order_hardcopy ,
   internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_number , request_order_hardcopy ,
   internal_tracking , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_status }
{ question_and_feedback_tracking , order_number , order_status ,
   product_returns }
{ question_and_feedback_tracking , order_number , order_status ,
   partner_tracking }
{ question_and_feedback_tracking , order_number , order_status ,
   partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_status ,
   internal_tracking }
{ question_and_feedback_tracking , order_number , order_status ,
   internal_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_status ,
   internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_number , order_status ,
   internal_tracking , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy }
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy , product_returns }
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy , partner_tracking }
```

```
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy , internal_tracking }
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy , internal_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy , internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_number , order_status ,
   request_order_hardcopy , internal_tracking , partner_tracking ,
   product_returns }
{ question_and_feedback_tracking , order_number , order_date }
{ question_and_feedback_tracking , order_number , order_date ,
   product_returns }
{ question_and_feedback_tracking , order_number , order_date ,
   partner_tracking }
{ question_and_feedback_tracking , order_number , order_date ,
   partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_date ,
   internal_tracking }
{ question_and_feedback_tracking , order_number , order_date ,
   internal_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_date ,
   internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_number , order_date ,
   internal_tracking , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy }
```

```
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy , product_returns }
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy , partner_tracking }
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy , internal_tracking }
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy , internal_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy , internal_tracking , partner_tracking }
{ question_and_feedback_tracking , order_number , order_date ,
   request_order_hardcopy , internal_tracking , partner_tracking ,
   product_returns }
{ question_and_feedback_tracking , order_number , order_date , order_status
    }
{ question_and_feedback_tracking , order_number , order_date , order_status
   , product_returns }
{ question_and_feedback_tracking , order_number , order_date , order_status
   , partner_tracking }
{ question_and_feedback_tracking , order_number , order_date , order_status
   , partner_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_date , order_status
   , internal_tracking }
{ question_and_feedback_tracking , order_number , order_date , order_status
   , internal_tracking , product_returns }
{ question_and_feedback_tracking , order_number , order_date , order_status
   , internal_tracking , partner_tracking }
```

```
{ question_and_feedback_tracking, order_number, order_date, order_status
   , internal_tracking, partner_tracking, product_returns }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy, product_returns }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy, partner_tracking }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy, partner_tracking, product_returns }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy, internal_tracking }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy, internal_tracking, product_returns }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy, internal_tracking, partner_tracking }
{ question_and_feedback_tracking, order_number, order_date, order_status
   , request_order_hardcopy, internal_tracking, partner_tracking,
   product_returns }
]
```

Listing D.1: Customer Service “*listProducts*” Result

Bibliography

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools.* Prentice Hall, 2nd edition, 2006.

[Alt10] Fadil Alturki. Jory: A tool for feature modelling based on product families algebra and bdds. Masters thesis, Department of Computing and Software, McMaster University, 2010.

[APS+10] A. Abele, Y. Papadopoulos, D. Servat, M. Törngren, and M. Weber. The cvm framework: A prototype tool for compositional variability management. *Proceeding of: Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, pages 101–105, 2010.

[AZKT10] Lamia Abo Zaid, Frederic Kleinermann, and Olga Troyer. Feature assembly: A new feature modeling technique. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson Woo, and Yair Wand, editors, *Conceptual Modeling - ER 2010*, volume 6412, pages 233–246. Springer Berlin Heidelberg, 2010.

[Bat05] D.S. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of SPLC'05*, pages 7–20, 2005.

[BCFH10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 159–162. University of Duisburg-Essen, 2010.

[BEL03] T. Bednasch, C. Endler, and M. Lang. Captainfeature tool. http://sourceforge.net/projects/captainfeature/, 2003. Last accessed on July 31, 2013.

[Beu08] D. Beuche. Modeling and building software product lines with pure::variants. In *Proceedings of SPLC'08*, page 358, 2008.

[BSTRc07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-cortés. Fama: tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Softwareintensive Systems VAMOS*, pages 129–134, 2007.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10:7–29, 2005.

[CPRS04] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. Xml-based feature modelling. In *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004*, pages 5–9. Springer-Verlag, 2004.

[EBB05] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 33–44, 2005.

[GFA98] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–85, Washington, DC, USA, 1998. IEEE Computer Society.

[HKM06] Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature algebra. In *In Formal Methods, volume 4085 of LNCS*, pages 300–315, 2006.

[HKM11] Peter Höfner, Ridha Khedri, and Bernhard Möller. An algebra of product families. *Software and Systems Modeling*, 10(2):161–182, May 2011.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. *Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University*, Novermber 1990.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. FORM: A feature-oriented reuse method with

domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[KL02] Kyo C. Kang and Jaejoon Lee. FOPLE - feature oriented product line software engineering: Principles and guidelines. *Pohang University of Science and Technolog*, 2002.

[KTS+09] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. Feature ide: a tool framework for feature-oriented software development. *Proceedings of ICSE'09*, pages 311–320, 2009.

[Lau06] Sean Quan Lau. Domain analysis of e-commerce systems using feature-based model templates. Masters thesis, University of Waterloo, 2006.

[Lev09] John Levine. *Flex & Bison*. O'Reilly Media, 2009.

[MBC09] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T. - software product lines online tool. In *Proceedings of OOPSLA'09*, pages 761–762, 2009.

[MC04] Antkiewicz M. and K. Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *The 2004 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse '04*. ACM Press, 2004.

[ML04] Thomas Maüen and Horst Lichter. RequiLine: A requirements engineering tool for software product lines. In Frank J. Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 168–180. Springer Berlin Heidelberg, 2004.

[Par76] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, 1976.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. *IDPT 2002*, 2002.

[Seb01] Robert Sebesta. *Concepts of Programming Languages*. Addison-Wesley, 2001.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edidtion edition, 2001.

[Str04] D. Streitferdt. *Family-Oriented Requirements Engineering*. Phd thesis, Technical University Ilmenau, 2004.

[vDK02] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology - CIT*, 10(1):1–17, March 2002.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 45–54. IEEE Computer Society, 2001.