

# **Jory: A Tool for Feature Modelling Based on Product Families Algebra and BDDs**

By  
FADIL ALTURKI, B.SC

A Thesis  
Submitted to the School of Graduate Studies  
in partial fulfilment of the requirements for the degree of  
Master of Applied Science in Software Engineering  
Department of Computing and Software  
McMaster University

© Copyright by Fadil Alturki, March 18, 2010

MASTER OF SCIENCE (2010)  
(Computer Science)

McMaster University  
Hamilton, Ontario

**TITLE: Jory: A Tool for Feature Modelling  
Based on Product Families Algebra and BDDs**

**AUTHOR:** Fadil Alturki, B.Sc (King Fahd University of Petroleum and Minerals)

**SUPERVISOR:** Dr. Ridha Khedri

**NUMBER OF PAGES:** xii, 110

# Abstract

Feature models are commonly used to capture the commonalities and the variability of product families. There are several feature modelling notations that correspondingly depict the concepts of feature modelling techniques. Therefore, the tools based on them reflect this diversity in the notations used and the fuzziness of the concepts adopted.

The thesis discusses the design and the construction of a tool that is based on Product-Families Algebra (PFA) and on Binary Decision Diagrams (BDD). The first brings the mathematical formalism to the specifications of product families and the mathematical theory that enables calculations on feature-models. The second brings efficient algorithms in time and in space. Hence, the tool allows several algebraic manipulations of feature models that are specified within the language of PFA. We coined this tool Jory.

The main contribution of the thesis is the design of the tool, and the implementation of four layers of its architectural design. As well, the thesis gives an implementation of multi-sets and the operations on them using BDDs.

Several case studies are presented and used in the validation of the tool.



# Contents

List of Figures	viii
List of Tables	x
<b>1 Introduction</b>	<b>1</b>
1.1 Problem . . . . .	6
1.2 Motivation . . . . .	7
1.3 Contribution . . . . .	8
1.4 Thesis Organisation . . . . .	10
<b>2 Feature Modelling Techniques and Tools</b>	<b>11</b>
2.1 Graphical Feature Modelling Techniques . . . . .	11
2.1.1 Convergence Between Feature Modelling Techniques . . . .	12
2.1.2 Divergence Between Feature Modelling Techniques . . . .	14
2.2 Non-graphical Feature Modelling Techniques . . . . .	16
2.2.1 Product Family Algebra . . . . .	17
2.3 Tools Supporting Feature Modelling . . . . .	24
<b>3 Product Family Algebra Implementation Using BDDs</b>	<b>27</b>
3.1 Introduction to Binary Decision Diagrams . . . . .	27
3.2 Implementing Product Family Algebra using BDDs . . . . .	30

3.2.1	The Implementation of the Set-Model . . . . .	30
3.2.2	The Implementation of the Bag-Model . . . . .	33
3.3	Conclusion . . . . .	42
<b>4</b>	<b>System Design</b>	<b>45</b>
4.1	Architectural Design . . . . .	45
4.2	Detailed Design . . . . .	47
4.2.1	The Interface Layer . . . . .	48
4.2.2	The Translation Layer . . . . .	49
4.2.3	The Term Evaluation Layer . . . . .	49
4.2.4	The Concrete Models Layer . . . . .	53
4.2.5	The BDD Layer . . . . .	54
4.3	Major Decisions, Capabilities, and Future Extensions . . . . .	60
<b>5</b>	<b>Testing and Validation</b>	<b>63</b>
5.1	Testing Techniques Adopted . . . . .	63
5.1.1	Unit Testing . . . . .	63
5.1.2	Integration Testing . . . . .	64
5.1.3	Parallel Testing . . . . .	65
5.1.4	Acceptance Testing . . . . .	65
5.2	Validation Examples . . . . .	66
5.2.1	The Employee Self Service . . . . .	66
5.2.2	The Robot Example . . . . .	70
5.2.3	Other Validation Remarks . . . . .	74
<b>6</b>	<b>Conclusion and Future Work</b>	<b>75</b>
6.1	Contribution . . . . .	76
6.2	Future Work . . . . .	77

<i>CONTENTS</i>	vii
A FM Techniques Notations and their corresponding PFA Expressions	79
B Robot Example Results	89





# List of Figures

1.1	Bicycle Feature Model in FODA . . . . .	3
2.1	Bicycle Feature Model in FODA . . . . .	14
2.2	AND-composition in FORM . . . . .	21
2.3	XOR-decomposition in FeatuRSEB . . . . .	21
2.4	OR-decomposition in FeatuRSEB . . . . .	21
2.5	Mandatory and Optional Features in PLUSS . . . . .	21
2.6	AND-Composition of a,b,c,d in FeatuRSEB . . . . .	22
2.7	XOR-Decomposition of e,f,g,h in FeatuRSEB . . . . .	22
3.1	The BDD of $f \triangleq x \vee \neg y \wedge z$ . . . . .	28
3.2	The BDD for the set $S$ . . . . .	29
3.3	The empty BDD . . . . .	31
3.4	The BDD corresponding to $P_1$ . . . . .	31
3.5	The BDD corresponding to $P_2$ . . . . .	32
3.6	The BDD corresponding to the product in $X$ . . . . .	32
3.7	The BDD corresponding to $P_w$ in $W$ . . . . .	34
3.8	The BDD corresponding to $P_z$ in $Z$ . . . . .	35
4.1	Architectural Design . . . . .	46
4.2	A Snapshot of the Interface of <i>Jory</i> . . . . .	48

4.3	The Term Evaluation Layer . . . . .	50
4.4	Concrete Models' Layer and the BDD Layer . . . . .	54

# List of Tables

A.1	FODA elements and the corresponding PFA expressions . . . . .	81
A.2	FORM elements and the corresponding PFA terms . . . . .	82
A.3	FeatuRSEB elements and the corresponding PFA terms . . . . .	83
A.4	van Gurp elements and the corresponding PFA terms . . . . .	84
A.5	Riebisch elements and the corresponding PFA terms . . . . .	85
A.6	PLUSS elements and the corresponding PFA terms . . . . .	86
A.7	Feature Modelling Notations and the Corresponding PFA Terms .	87



# Chapter 1

## Introduction

This thesis is on the design and the implementation of a tool for *product families*. The tool implements an algebraic *feature modelling technique* that is based on *Product Families Algebra* (PFA) [HKM09]. The thesis aims to extend the benefits of the various available feature modelling techniques used for handling product families, and empower them with mathematics for specification, analysis, calculus, and inference.

In this chapter, we introduce the basic concepts related to this work and illustrate them with simple examples. We introduce feature modelling using FODA [KCH<sup>+</sup>90] - a graphical feature modelling technique. We then present feature modelling using PFA. We follow with the problem we need to tackle, the motivation, and the contribution of this work.

In the automotive industry, cars are manufactured in product lines. A year's model, can be seen as a *product family*. The members of the car family are built from basic components or artefacts that are sometimes called *features*. Usually, the members of this family have one or more prototype family members from which they are derived. A prototype member contains the aggregation of what is called the *commonality* features. The products are differentiated by what is

referred to as the *variability* features. In the car product family, all the members have engines, transmission systems, and many other *common* features. However, some of these cars (products) derived from a prototype, are distinguished by having powered windows and doors. Yet, other cars are distinguished by having advanced audio systems and GPS systems.

The concept of *product families* is adapted in Software Engineering. It is sometimes referred to as *Software Product Lines*(SPL) [KDn06]. A software family example can be a set of editions of an operating system. The set of operating systems can be modelled as a family, where we have editions for personal computers; for those with 32-bit and those with 64-bit architectures. We can also have editions for servers with 32-bit and 64-bit architectures, and editions for netbooks and older machines. One way to analyse and build such a hierarchy is using Feature Modelling.

*Feature Modelling* (FM) is an approach to capturing software families in terms of features. The outcome of the process of feature modelling is a *feature model*. There are several FM techniques and most of them produce graphical feature models. One FM technique is FODA which was introduced in 1990 [KCH<sup>+</sup>90]. There are other FM techniques used like FORM [KKL<sup>+</sup>], FeatuRSEB [GFd98], Generative Programming [Cza98] (GP) and PLUSS [EBB]. For the graphical FM techniques, a feature model is a graphical representation of a product family where the vertices correspond to the basic or composite features and the edges corresponds to the relationships between them.

Consider a factory producing bicycles. Looking at the basic parts of a bicycle, we see that it consists of a frame set, a wheels set, a front set, a saddle set, gears, pedals, and brakes. Every part of these consists of a set of components as laid out in the FODA feature model of our bicycle product family given Figure 2.1.

In FODA, we call the root node, a *concept* and we enclose it in a box. The

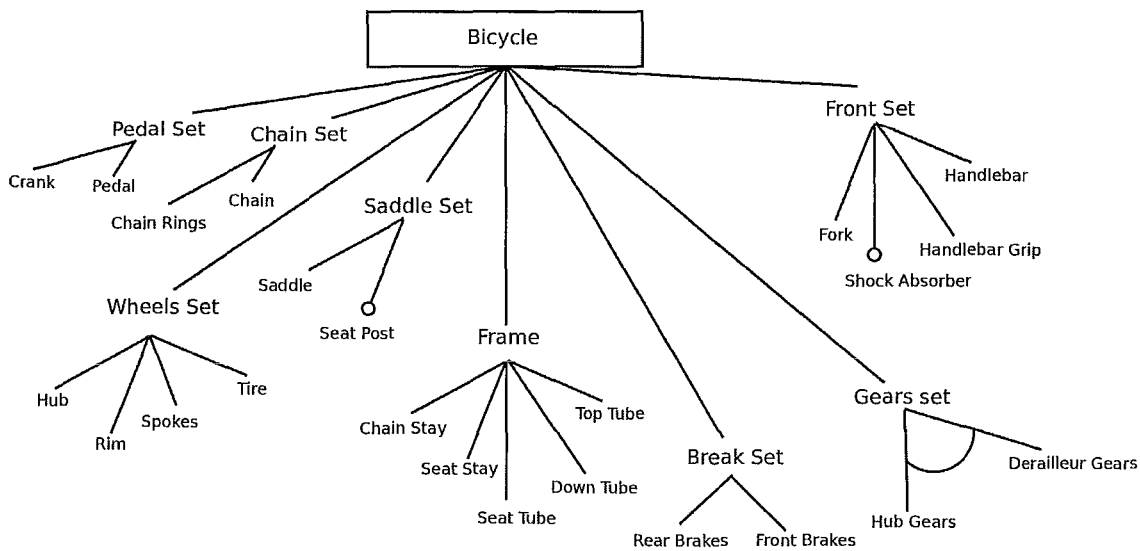


Figure 1.1: Bicycle Feature Model in FODA

concept node is the *bicycle* in our case. Every node is a feature. A feature can be primitive or compound. The *top\_tube* is a primitive feature while the *frame* is compound. We denote that the frame is composed of the features underneath (the different tubes) by using the AND-composition notation. It is denoted by the edges coming out of the frame node. When we have alternatives, we use the XOR-composition like in the case of the gears. The gear system selected has to be one and only one of the two types of gears. The XOR is denoted in a similar way to the AND but with an arc crossing the edges.

A feature can be *mandatory* or *optional*. The *saddle* is a mandatory feature while the *seat\_post* is optional. We show that a feature is optional using a blank circle on the node. Constraints exist in feature models. In some cases, we will need to state that a feature *requires* another feature or a feature is *mutually exclusive* with another. In FODA, constraints are expressed in text.

FODA has been adapted and extended by other FM techniques [SHTB07]. There are several graphical notations introduced to accommodate new concepts and semantics. FeatuRSEB supports the OR relation. In FORM and FO-PL [KLD02] techniques, features are depicted inside boxes. van Gurp [vGBS01] technique represents features with different box styles to express internal features and external features. Riebisch [RBSP02] technique adapts use-case terminologies and UML notations. GP takes into the consideration the number of occurrences of a feature in a family and grouping features and assigning an occurrence to them. While FODA, FORM and GP state the constraints in text, van Gurp and Riebisch express them graphically using dashed lines with labels. In Chapter 2, we provide a thorough survey of the literature of the FM techniques.

There are several attempts to extend these techniques in another perspective. The above extensions are concerned with the specification of product families using graphical notations. But there is always a need to express complex constraints. For example, we can have a constraint which states that in a product, the selection of a feature requires the selection of another feature to exist within a range of the number of occurrences. In the cases of such constraints, we sometimes need to verify whether a product considered faulty or undesirable. Such needs lead to introduce propositional logic into the context [Bat05] and the use of Prolog in *Pure::Variants* [Pur10]. We also need to perform validations and calculations. We need to know the answers to questions like, how many potential products of bicycles based on the feature model we have are there, or how similar two products are.

The above FM techniques are graphical. As far as we know, there is only



one non-graphical FM technique which is Product Families Algebra (PFA). One of PFA aims is to provide a means for the specification of product families algebraically. This extends the feature modelling with calculus, analysis, and inference.

PFA is an algebraic structure, that is an idempotent commutative semiring  $(S, \cdot, +, 0, 1)$ . We formally present PFA in Chapter 2. Intuitively, we say,  $S$  is the set of all the product families,  $\cdot$  is a binary operation interpreted as composition or mandatory presence,  $+$  is a binary operation interpreted as choice,  $0$  is a faulty product family, and  $1$  is the empty family.

We can specify the *pedal\_set* in PFA as  $\text{pedal\_set} = \text{pedal} \cdot \text{crank}$ . This states that the *pedal\_set* is composed of a *pedal* and a *crank*. These two features are mandatory to every *pedal\_set*.

We can specify the *saddle\_set* as  $\text{saddle\_set} = \text{saddle} \cdot (\text{seat\_post} + 1)$ . This states that the *saddle\_set* is composed of a mandatory *saddle* and an optional *seat\_post*. We express that a feature  $f$  is optional using the plus  $+$  and  $1$ , and we write  $(f + 1)$ ; the choice of the feature or nothing. For the *gears\_set*, we write  $\text{gears\_set} = \text{hub\_gears} + \text{derrailleur\_gears}$ . This states that we can choose either one of the gear sets, but not both at the same time (i.e., it gives a way to write *XOR*).

We introduce PFA formally in Chapter 2 with larger examples and show more capabilities like product refinements and constraints.

With PFA, we can also do calculus on features and products. We can know the potential number of possible products, the number of products that have a combination of features but not another combination. In addition, we can

validate product families, simplify them, and combine them. We can infer information from the specification such as whether a family is a subfamily of another, or whether a product refines another.

For the support and feature modelling, several tools were implemented. However, they are mostly concerned with the graphical FM techniques. Some of the tools are: *AmmiEdi* [Gen10], *CaptainFeature* [Sou10], *FeaturePlugin* [AC04], *Pure::Variants* [Pur10], *RequiLine* [vdML04], and *XFeature* [XFe10]. For PFA, there is only a prototype tool implemented in Haskell [HKM06].

## 1.1 Problem

Feature modelling tools support mostly one FM technique or their proprietary notations [DSF07]. They do not have a way to convert from one notation to another. In some cases, they add some extensions and new concepts that are not part of the supported technique or not part of feature modelling concepts like what we found in *Pure::Variants* [Pur10] and *RequiLine* [Req10]. Some of the tools are not fully dedicated to feature modelling but feature modelling is a sub-functionality of a CASE tool like CASE-FX [FAC07].

Furthermore, feature models have the advantage of the visualization of product families. However, the visualization feature is by itself a problem for large systems and hard to utilize for calculation, analysis and inference.

PFA is a formal FM technique that extends feature modelling with mathematical capabilities. Our work intends to provide the design and the implementation

of a tool based on PFA that contribute to feature modelling and extend the other FM techniques and their benefits.

## 1.2 Motivation

PFA defines the concepts of feature modelling mathematically for the purpose of handling product families precisely and rigorously. For instance, it explicitly defines the terms *product*, *family* and *feature* in the context of product families. This helps avoid the ambiguity found in the graphical FM techniques where features sometimes are primitive, and sometimes are compound.

PFA is an idempotent commutative semiring. It inherits all the benefits of this algebraic structure and the benefits of the mathematical models it implements such as sets and bags. Furthermore, PFA provides compact specifications of product families - we refer the reader to Section 2.2.1. With this and the above benefits, it can be used for large and critical systems. Given the specification is a set of algebraic formulas, they can be manipulated and processed mathematically.

PFA can also extend the graphical FM techniques. A feature model can be translated from its graphical notation to the language of PFA. We can then simplify the specification, factorize it, merge it with another before translating the result into a graphical feature model. In this case, we can also search for a product family, count the number of products satisfying certain conditions, or find the size of a certain family.

As we can translate from a graph notation to the PFA language, we should

be able to translate from an FM technique notation to another via PFA. We should be able to translate from FODA to PLUSS for example. This makes PFA work as a bridge between the various FM techniques.

For the problem of the visualization of large product families, we can extract sub-graphs of interest from a feature model, encoded in PFA, and show them in a given FM technique notation. We should be able to customise the way we display the feature model through grouping and selected product families relationships.

The tool is not only an implementation of PFA and its basic operations, but it can be an all-in-one platform that facilitates feature modelling and brings the benefits of FM techniques together in one place.

## 1.3 Contribution

This thesis provides the design and implementation of the tool *Jory* for feature modelling based on Product Family Algebra. The design aims to establish a platform for bringing the benefits of feature modelling whether they are graphical or non-graphical techniques in one place.

We have designed the tool in a layered architecture with separation of concerns in mind. The layers can be managed, maintained and replaced independently. We take into consideration the need to scale and extend the functionality and incorporate new future concepts and techniques. We also aim to make the tool cross-platform and serve all the audiences in the community of feature modelling.

To handle large systems efficiently, we have implemented PFA using *Binary Decision Diagrams* (BDDs). We implemented PFA for the set model and the bag model in BDDs. Using BDDs, we encode the features in binary diagrams and we use the efficient algorithms to perform PFA operations.

We parse and evaluate the user input of the specification of the product family, and produce terms that can be then manipulated mathematically. The terms and expressions generated can be then calculated, validated, analysed and queried. We can then extract features, products, families, apply constraints, and make assessments.

Using syntax-based translation, feature models can be translated from graph notations to the language of PFA for processing or translation to other feature models. Through this translation, visualizations can be customised. It can ease the communication between groups that are using different FM techniques.

We have implemented the core part of the tool. We have implemented PFA using BDDs with two models: a set model and a bag model. The bag model is implemented to handle product families when feature duplication is required. The tool is now capable of taking specifications of product families and is capable of parsing and processing them. It can perform calculus, analysis and inference on product families. The designed tool is seen as a a mathematically-based platform for feature modelling bringing all the benefits of the current state of the art techniques and practices, and extendible to new concepts and ideas.

## 1.4 Thesis Organisation

In Chapter 2, we take a closer look at the graphical and the non-graphical FM techniques and introduce PFA and its mathematical background. In Chapter 3, we discuss the implementation of PFA using BDDs. In Chapter 4, we give the architectural and the detailed design of the tool. The validation procedures and the test results are given in Chapter 5. In Chapter 6, we conclude and point to future work.

# Chapter 2

## Feature Modelling Techniques and Tools

In Chapter 1, we introduced with simple examples, the basic concepts of feature modelling. In this chapter, we explore the FM techniques and the tools we found in the literature. We first study the graphical FM techniques and illustrate their similarities and differences with examples. We then follow with non-graphical FM techniques and present PFA in detail with illustrations. We conclude the chapter with an overview of the tools found in the literature.

### 2.1 Graphical Feature Modelling Techniques

In our survey of the literature, we found the following techniques: FODA [KCH<sup>+</sup>90], FORM [KKL<sup>+</sup>], FOPLE [KLD02], FeatuRSEB [GFd98], Generative Programming [Cza98] (GP), FORE [Str04], Riebisch Technique [RBSP02], van Gurp Technique [vGBS01] and PLUSS [EBB].

To have a good understanding of the graphical techniques, we first introduce

the terminology where they converge and follow that with where they diverge. In Appendix A, we provide tables illustrating every FM technique notation.

For the comparison of the terminologies, concepts and graphical notations, we refer to the FM techniques papers and to [DS06], [Kot05], [Rob03] and [SHTB07].

### 2.1.1 Convergence Between Feature Modelling Techniques

Looking at where they converge, we observe that all the FM techniques share the concept of *feature* and *feature model*. They all use graphs (nodes and edges) to represent their feature models. The nodes are the features and the edges are the relations between these features. The term “feature” is ambiguous. It refers to nodes. A node can be primitive, compound, a root or a leaf. The root node is called a “concept” where it represents a set of features. Internal nodes might not be primitive. A node, which is still called a “feature”, can be a combination of other subsequent features.

Figure 2.1 gives the feature model in FODA of the bicycle example. The *concept* node is the *bicycle* and it is the root feature. The nodes are features, related by edges. The *frame* is a compound feature and the *fork* is primitive.

For the perspective of nodes, they all have the following concepts of:

- *mandatory*: a feature is mandatory if it must be selected in a product.
- *optional*: a feature is optional if can be selected in a product.

In our example of Figure 2.1, the *break\_set* is a mandatory feature while the *shock\_absorber* is optional.

For the edges, all the FM techniques, except for Riebisch’s, have the following concepts of:



- *XOR-decomposition*: a relation from a feature  $x$  to a set of other features such that  $x$  is composed of only one of the features of that set. It is sometimes used to denote *alternative* features. In this case, a feature is selected from a set of alternative features i.e. one and only one feature must be selected.
- *AND-composition*: a relation from a feature  $x$  to a set of other features such that  $x$  is composed of *all* of the features of that set. In this case, this node, or feature, is compound.

In Figure 2.1, the *gears\_set* is an *XOR-decomposition* of *hub\_gears* and *derailleur\_gears* feature while the *shock\_absorber* is optional. In this case, we have to choose one of these features but not both at the same time for a *gears\_set*. An example of an *AND-composition* is *chain\_set* where it is composed of *chain* and *chain\_rings*.

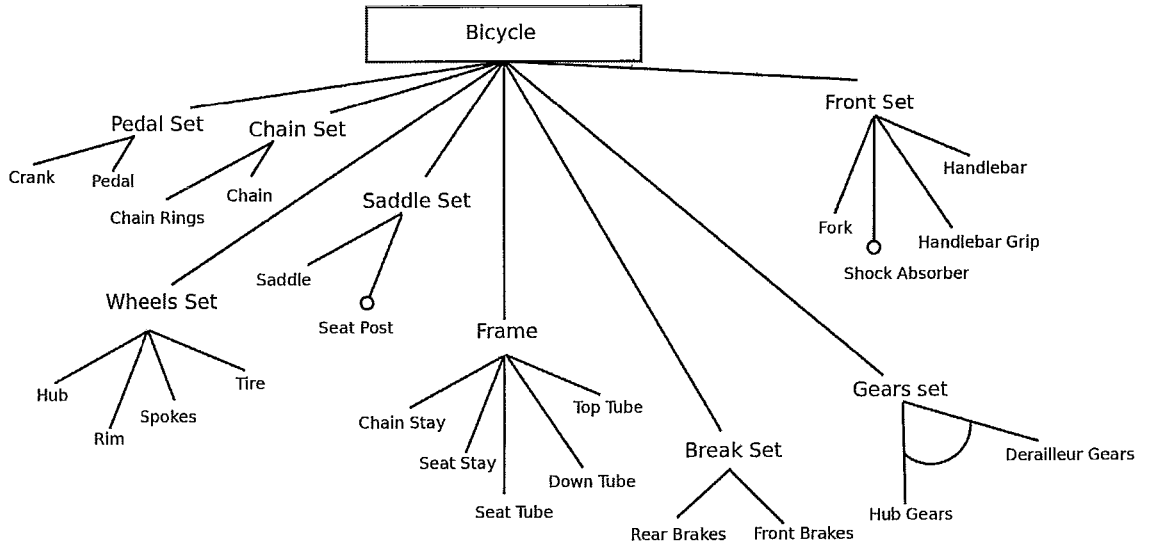


Figure 2.1: Bicycle Feature Model in FODA

All the surveyed feature model techniques denote the dependencies between features using *constraints*. Constraints could be on the following:

- *requires* constraint: it states that the selection of a feature  $x$  *requires* a feature  $y$ .
- *mutex* or *excludes* constraint: it states that the selection of a feature  $x$  is *mutually exclusive* with the selection of a feature  $y$ . Either  $x$  is selected or  $y$  but not both at the same time.

### 2.1.2 Divergence Between Feature Modelling Techniques

However, these FM techniques diverge in many ways. We present this diversion with regards to the following perspectives: the overall feature model graphs, nodes, edges and constraints.

From the perspective of the nature of graphs, the diagrams of FODA, FO-  
PLE, GP and PLUSS are trees. On the other hand, the diagrams of FORM,

FeatuRSEB, FORE, Riebisch's and van Gorp's technique are directed acyclic graphs (DAGs).

From the perspective of nodes, in FOPLE and FORM, the nodes are depicted in boxes. On all other FM techniques, they are depicted as textual names and only the concept node is represented as a box. All FM techniques represent the concept of *mandatory* and *optional* using nodes, except Riebisch's technique which uses edges, instead. All other feature models - those using nodes - use circles to represent optional features. Optional features are represented by blank circles. Riebisch's technique and PLUSS use blank circles for optional features and black circles for mandatory ones.

From the perspective of edges, most of the FM techniques represent *XOR* and *OR* using edges except Riebisch's technique and PLUSS. Some techniques, like FODA and FORM, do not have the *OR*. Riebisch's technique uses UML-like multiplicities and PLUSS uses the concept of *single adapter* for *XOR* and *multiple adapter* for *OR*.

For the constraints, FODA and FORM have their constraints written in text. On the other hand, FeatuRSEB and other techniques represent them as textual and graphical dashed arrows.

For the the concept of number of the occurrences of features, it is used only in GP and Riebisch's techniques.

In van Gorp technique, there are some more differences to note. We find the following concepts:

- *binding times* features: represented by solid-line boxes. In [vGBS01], the concept "binding times" is used to express the need for a delayed decision. It takes into consideration the feature aggregations that happen at different variation points of the software life cycle and these points are called "binding times". We are concerned with this the time where a product is selected.

- *external* features: represented by dashed-line boxes. The “external” features are those that are external to the system modelled and part of the target platform. These are considered in van Gurp’s technique because the modelled family depends on them.
- *XOR-decomposition*: represented by a blank triangle.
- *and-composition*: represented by a black triangle.

In GP, there is the concept of the *group* of features and the concept of a group cardinality. In GP, we can put a set of features in group and assign it a minimum and maximum cardinality. This implies that for the selection, the number of features selected in this group, should be in the inclusive range between the minimum and the maximum.

The graphical and the textual notations of feature models, are illustrated in Appendix A. In addition to notation illustrations and the usage, we also write the corresponding PFA expressions.

## 2.2 Non-graphical Feature Modelling Techniques

There are several attempts to do feature modelling using means other than graphs. In [DKDK02], they introduce a feature modelling language (FDL) which is a textual language to describe features. In [HM85], they introduce a minimal set of algebraic laws claimed to be sound and complete to be used for feature modelling refactoring. [Bat05] combines feature models with propositional logic to debug them using satisfiability solvers. There is also the attempt

in [CHE05] where they specify feature models as context-free grammars. They transform the language recognized by grammars and specify cardinality-based feature models. There is also the attempts to express the constraints in Prolog as in *Pure::Variants*, or express them in tool-specific languages.

Another attempt is Product Family Algebra (PFA). PFA is an idempotent commutative semiring devised to specify feature models. The theory derived from PFA enables calculus and inference on product families. PFA can handle the specifications, constraints, view reconciliation and other feature modelling needs.

We choose this all-in-one technique for our tool as a kernel that can work well with feature models, translations, transformations, calculus, and inference. In the next section, we give an overview PFA and show how the various FM notations are expressed in PFA terms.

### 2.2.1 Product Family Algebra

Product Family Algebra (PFA) was proposed to provide a formalism in which feature modelling terms are defined precisely and software families are dealt with mathematically. In this section, we define Product Family Algebra and introduce its concepts and how it relates to the graphical FM techniques. The material in this section is mainly from [HKM06, HKM08, HKM09].

**Definition 2.2.1.** ([HKM06]) A *semiring* is a quintuple  $(S, +, 0, \cdot, 1)$  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, i.e.,  $0 \cdot a = 0 = a \cdot 0$ . The semiring is *commutative* if  $\cdot$  is commutative and it is *idempotent* if  $+$  is idempotent, i.e.,  $a + a = a$ . In the latter case, the relation  $a \leq b \stackrel{\text{def}}{\Leftrightarrow} a + b = b$  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  $\leq$ .  $\square$

Product Family Algebra (PFA) is a mathematical algebraic structure that is an idempotent commutative semiring.

For PFA, the universe of discourse is the *collection* of families. A *product* is made up of an aggregation of features and a *product family* is a collection of products. The operator  $+$  can be interpreted as the choice operator between product families. On the other hand, the operator  $\cdot$  can be interpreted as the composition operator of product families. The  $+$  can be used to express *optional* features and the  $\cdot$  can be used to express *mandatory* features. With these two operators, PFA can express the relationships found in the feature modelling techniques.

We go through the core concepts of PFA and illustrate with simple examples. We start by defining what is a *product family*, a *product*, a *feature* and *refinement*.

**Definition 2.2.2.** ([HKM08]) A product family  $a$  is a **product** if

$$\forall(b \mid: b \leq a \Rightarrow b = 0 \vee b = a) \text{ and}$$

$$\forall(b, c \mid: a \leq b + c \Rightarrow a \leq b \vee a \leq c) .$$

In particular,  $0$  is a product. A product  $a$  is **proper** if  $a \neq 0$   $\square$

**Definition 2.2.3.** ([HKM08]) An element  $a$  is called **feature** if it is a **proper product** and

$$\forall(b \mid: b|a \Rightarrow b = 1 \vee b = a) \text{ and}$$

$$\forall(b, c \mid: a|(b \cdot c) \Rightarrow (a|b \vee a|c)) .$$

where the divisibility relation  $|$  is given by  $x|y \iff \exists z : y = x \cdot z$ .  $\square$

We say that this algebra is *feature-generated* if every element we have, is a finite sum of finite products of features. In this case, the *size* of element  $a$  is the minimum number  $n$  such that  $a = \sum_{i \leq n} p_i$  for suitable products  $p_i$ .

**Definition 2.2.4.** ([HKM06])

The **refinement relation**  $\sqsubseteq$  on a product families algebra is defined as

$$a \sqsubseteq b \iff \exists(c \mid c \in S : a \leq b \cdot c)$$

where  $\leq$  is the natural order on  $S$ . □

Informally, we can say that a product family  $a$  is said to *refine* another product family  $b$  if  $a$  has the same set of features and more. This is expressed as  $a \sqsubseteq b$ . Intuitively, we say that the product  $a$  has the same features of  $b$  or more. And we say that a family  $a$  *refines* another family  $b$  to mean that *all* the members of  $a$  refine *some* of the members of  $b$ .

There are several models that can be given for PFA. Two useful models are the *set model* and the *bag model*. We use the *set model* when our product families are not concerned with multiple occurrences of features and we use the *bag model* otherwise.

**2.2.1.1 The Set-based Model**

Let  $\mathbb{F}$  be a set of *features*. Then the set of all possible products is given by  $\mathbb{P} \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{F})$ . We call the *set of features*, as in Definition 2.2.2, a *product* and we call the *set of products* (an element of  $\mathcal{P}(\mathbb{P})$ ) a *product family*. The 0 in this model is defined as  $\{\}$  or  $\emptyset$ , and the 1 is defined as  $\{\{\}\}$  or  $\{\emptyset\}$ .

The operation  $\cdot$  on product families expresses the composition of features.

$$\begin{aligned} \cdot : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) &\rightarrow \mathcal{P}(\mathbb{P}) \\ P \cdot Q &= \{p \cup q : p \in P, q \in Q\}. \end{aligned}$$

On the other hand, the operation  $+$  expresses the choice between products in

different product families.

$$\begin{aligned} + : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) &\rightarrow \mathcal{P}(\mathbb{P}) \\ P + Q &= P \cup Q. \end{aligned}$$

### 2.2.1.2 The Bag-based Model

If we need to handle multiple occurrences of features in a product, then we should adapt multi-sets (bags) of features which is denoted by IPFB.

In this model, the 0 and the operation  $+$  are in the same way as those of the *set model*. The 1 is defined as  $\{\{\}\}$  or  $\{\emptyset_B\}$ .

However, we define the operation  $\cdot$  on product families as the composition of features.

$$\begin{aligned} \cdot : \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) &\rightarrow \mathcal{P}(\mathbb{P}) \\ P \cdot Q &= \{p \sqcup q : p \in P, q \in Q\} \end{aligned}$$

where  $\sqcup$  is the bag union. This is how we take into consideration the occurrences of features in the *bag model.XOR*.

Now, we give illustrative examples expressed in PFA terms and see how they relate to the graphical feature modelling. Figure 2.2 is given using *FORM* notation. This is an *AND-composition* of  $b, c$  and  $d$ . If we take this in the *set model*, we express this in PFA with the sentence  $a = b \cdot c \cdot d$ . It states that the features in the product  $a$  are *mandatory*, which is implied by the use of the  $\cdot$  operation.

Figure 2.3 is an *XOR-decomposition* of the features  $b, c$  and  $d$  in FeaturSEB. To express this in PFA, we write  $a = b + c + d$ . The  $+$  operation in this expression implies that these are *alternatives*. To instantiate a *product* from the *product family*  $a$ , only one of  $b, c$  and  $d$  should be selected.

A more interesting example is the *OR-decomposition* of  $b, c$  and  $d$  in FeaturSEB in Figure 2.4. We write this in PFA as  $a = b + c + d + (b.c) + (b.d) +$



$(c.d) + (b.c.d)$ .

Figure 2.5 is an illustration of a feature  $b$  that is *optional* and a feature  $c$  that is *mandatory*. To express this in PFA, we write  $a = (b + 1).c$ . The  $+$  operation between  $b$  and 1 implies that the *feature*  $b$  is optional.

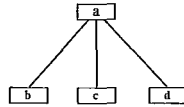


Figure 2.2: AND-composition in FORM

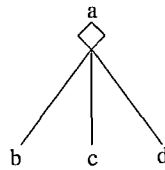


Figure 2.3: XOR-decomposition in FeatuRSEB

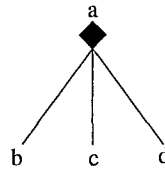


Figure 2.4: OR-decomposition in FeatuRSEB

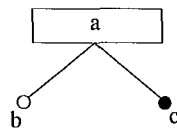


Figure 2.5: Mandatory and Optional Features in PLUSS

To see how we model product families using PFA, we take two families in Figure 2.6 and Figure 2.7 which are given in FeatuRSEB notation. Assume that our model is the *set model*. The product family in Figure 2.6 can be written in PFA as  $a = b \cdot c \cdot d$ . The product family in Figure 2.7 can be written in PFA as

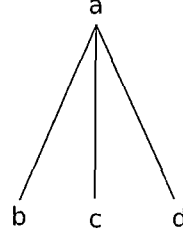


Figure 2.6: AND-Composition of a,b,c,d in FeatuRSEB

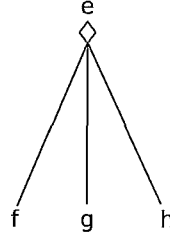


Figure 2.7: XOR-Decomposition of e,f,g,h in FeatuRSEB

$e = f + g + h$ . To express that fact that  $p$  is a product family composed of  $a$  and  $e$ . We write this as  $p = a \cdot e$ .

We can express  $a$  in set notation as  $\{\{b, c, d\}\}$  and  $e$  as  $\{\{f\}, \{g\}, \{h\}\}$ . The product family  $p$  is then  $\{\{b, c, d, f\}, \{b, c, d, g\}, \{b, c, d, h\}\}$ . For the product family  $p$ , the number of products are three. The family  $p$  contains the products  $\{b, c, d, h\}$ ,  $\{b, c, d, g\}$ , and  $\{b, c, d, f\}$ . The commonality features in  $p$  are in  $\{b, c, d\}$ . The family  $a$  is not a sub-family of  $p$ , but  $p$  refines  $a$ . In a similar way, we can perform more complicated calculus, analysis, and inference on product families.

To illustrate the refinement relation in the *set-model*, let  $f_1$  be a family containing two products where  $f_1 = \{\{a, b, c\}, \{d, e, f\}\}$ . Moreover, let  $f_2 = \{\{a, b\}, \{e, f\}, \{g, h\}\}$ . We say that  $f_1$  *refines*  $f_2$  and we write  $f_1 \sqsubseteq f_2$ . Intuitively, this indicates that every product in the family  $f_1$  refines some of the

products in the family  $f_2$ . We prove this using the definition of the refinement relation 2.2.4 as follows.

We need to show that

$$\begin{aligned}
& f_1 \sqsubseteq f_2 \\
\iff & \quad \langle \text{By definition of refinement 2.2.4} \rangle \\
& \exists(f_3 \mid f_1 \leq f_2 \cdot f_3) \\
\iff & \quad \langle \text{using } a \leq b \stackrel{\text{def}}{\iff} a + b = b \rangle \\
& \exists(f_3 \mid f_1 + f_2 \cdot f_3 = f_2 \cdot f_3) \\
\iff & \quad \langle \text{set } f_3 = \{\{c\}, \{d\}\} \text{ and substituting with the sets corresponding} \\
& \quad \text{to } f_1 \text{ and } f_2 \rangle \\
& \{\{a, b, c\}, \{d, e, f\}\} + \{\{a, b\}, \{e, f\}, \{g, h\}\} \cdot \{\{c\}, \{d\}\} = \\
& \{\{a, b\}, \{e, f\}, \{g, h\}\} \cdot \{\{c\}, \{d\}\} \\
\iff & \quad \langle \text{multiplying } f_2 \text{ and } f_3 \rangle \\
& \{\{a, b, c\}, \{d, e, f\}\} + \{\{a, b, c\}, \{c, e, f\}, \{c, g, h\}, \{a, b, d\}, \{d, e, f\}, \{d, g, h\}\} = \\
& \{\{a, b, c\}, \{c, e, f\}, \{c, g, h\}, \{a, b, d\}, \{d, e, f\}, \{d, g, h\}\} \\
\iff & \quad \langle \text{adding } f_1 \text{ to } f_2 \cdot f_3 \rangle \\
& \{\{a, b, c\}, \{c, e, f\}, \{c, g, h\}, \{a, b, d\}, \{d, e, f\}, \{d, g, h\}\} = \\
& \{\{a, b, c\}, \{c, e, f\}, \{c, g, h\}, \{a, b, d\}, \{d, e, f\}, \{d, g, h\}\} \\
\iff & \\
& \text{true}
\end{aligned}$$

To handle multiple occurrences of features, we can use the bag model. If we want to have a product that is composed of  $b, c$  and  $d$  with 3 occurrences of  $b$ , we write  $a = b.b.b.c.d$ . All other expressions used in the graphical FM techniques can

be expressed in PFA such *requires*, *mutex* and external features. In Appendix A, we include a table for each FM technique that uses a new notation or a concept. We invite the reader extend on this with a wide range of theory on PFA and examples which are found in [HKM06, HKM08, HKM09].

PFA inherits all the mathematics of idempotent semirings and provides us with a calculational power that can be applied to graphical feature models. It also inherits all the mathematics of sets and bags or any models for PFA. With this in hand, we can do all sorts of calculations and inferences using one that has all-in-one capabilities. A feature model can be, for instance, simplified, factorized, and mathematically analysed. Moreover, feature models can be combined, extracted, excluded, and validated. We conjecture that this should provide more functionality that can be applied to feature models to extend their benefits.

## 2.3 Tools Supporting Feature Modelling

In the literature, there are several feature modelling and FM techniques tools. In [DSF07], we find an industry survey of the tools and their evaluation. The survey covers the following tools: *Captain Feature* [Sou10], *Pure::Variants* [Pur10], *Feature Plugin* [AC04], *SSEP toolset* [SSP<sup>+</sup>00], *DecisionKing* [DGR07], *DOORS T-REK* [IBM10], *XFeature* [XFe10], *FMP* [CAK<sup>+</sup>05], *FORM/ASADAL* [KSA<sup>+</sup>06], *Gears* [Big10], *Var-Mod* [Uni10], and *RequiLine* [Req10]. We have also found other tools in the literature like *001* [Kru93], *DOMÉ* [Hon10] and *CASE-FX* [FAC07].

Most of these tools do not support the classical feature modelling techniques. A few tools were built to support a single FM technique such as *AmiEddi* which supports mainly *GP*. Most of the tools support feature modelling as a partial functionality of the tool. An example of tools that are not fully dedicated to

feature modelling but support it as a sub-functionality of a CASE tool is *CASE-FX* [FAC07]. The tools included in the survey above support FODA-like notations with some tool-specific language and extensions. Although these tools share some background, they do not produce equivalent feature models. They do not support the conversion or the translation from one technique to another. If a user lacks a feature modelling capability in one technique, the user has to switch to another tool that supports it. In some cases, the tools add some extensions and new concepts that are not part of the supported techniques or part of feature modelling concepts like what we found in *Pure::Variants* and *RequiLine*. The tool *001* has a set of utilities part of which is feature modelling. The feature models are stored as *001 TMap* as an extension to FODA notation. *DOME* uses UML-like notations. There are several attempts to enhance the way we handle feature models and analyse them. *Requiline* uses an Oracle/SQL database *Pure::Variant* uses Prolog for the constraints.

In this work, we intend to take feature modelling a step further. We intend to base the feature model techniques on a platform based on PFA. Our aim is to make PFA as a bridge to translate from one technique to another. Moreover, PFA will be used for formal specification, analysis, inference, and generation of feature models. In the next chapter, we explain how we implement PFA using Binary Decision Diagrams.



# Chapter 3

## Product Family Algebra Implementation Using BDDs

In Chapters 1 and 2, we introduced Feature Modelling, the graphical and the non-graphical techniques, and their existing supporting tools. We introduced Product Family Algebra and illustrated how PFA can extend the benefits of Feature Modelling. In this chapter, we show how we implemented two models of Product Family Algebra using Binary Decision Diagrams (BDDs). We first introduce BDDs and present their main benefits. We then show how BDDs can be used to implement PFA. We then explore some examples to show how our implementation works.

### 3.1 Introduction to Binary Decision Diagrams

Binary Decision Diagrams are directed acyclic graphs (DAGs). They are compact, yet efficient means of representing boolean functions. A BDD is a binary tree where the nodes represent the boolean variables and the terminal nodes are either 0 or 1. Each node has two child nodes, one is connected to by a low edge, and

another is connected to by a high edge. We say that a node/variable is selected, if we select the high edge, i.e. the boolean value is 1 and 0 otherwise.

If we take an example of a boolean function  $f \triangleq x \vee \neg y \wedge z$ , then we can encode all the possible values of  $f$  in a truth table with all the possible values of  $x, y$  and  $z$ . That can grow exponentially as the number of variables increase. We can represent the same function in a compact graph, i.e. using a BDD. We can then analyse the behaviour  $f$  and calculate the possible values using the graph with no need to decompression. The BDD for the above function of  $f$  is given in Figure 3.1. We present the high edges with solid lines and the low edges with dotted lines. In this BDD, we see that the function  $f$  is true, if  $x$  is selected alone. Also, it is true when we select *not*  $y$  and  $z$  at the same time.

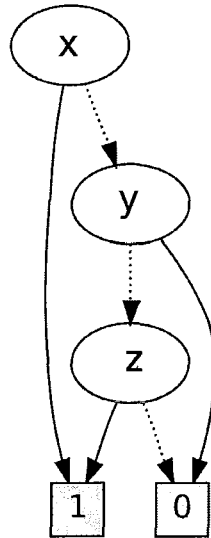


Figure 3.1: The BDD of  $f \triangleq x \vee \neg y \wedge z$

We are actually using Reduced Ordered Binary Decision Diagrams (ROBDDs) [Bry92]. In ROBDDs, the nodes are always ordered. In addition, we remove the redundancy of sub-graphs and edges when a node has its high and low edges point to the same destination. In ROBDDs, the nodes are unique; no two nodes have the same label in different levels. For simplicity, we call ROBDDs,



BDDs.

We can also use BDDs to represent sets. Let  $S \triangleq \{a, b, c, d\}$  and let the universe of discourse  $U \triangleq \{a, b, c, d, e\}$ . The set  $S$  can be represented using a BDD by simply indicating those elements in the universe of discourse  $U$  from which  $S$  is constructed belong to the set. Those which do not belong to  $S$ , do not show in the BDD. The BDD in Figure 3.2 depicts the set  $S$  that contains the elements  $a, b, c, d$ . We can read that as  $a, b, c, d \in S$ .

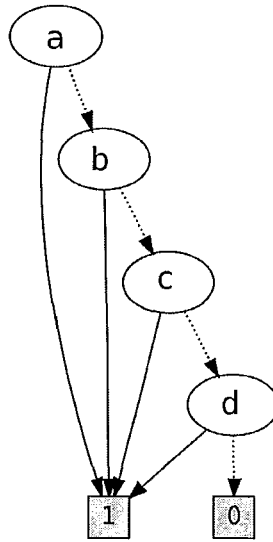


Figure 3.2: The BDD for the set  $S$

In the literature, there are many known and useful algorithms on BDDs [Bry86]. The algorithms provide efficient operations on boolean functions with large number of variables. The algorithms include reductions, simplifications, application of logical operations on BDDs like *and* and *or*, restricting BDDs by a value of a variable or a sub-BDD, checking for existence of a variable or an expression in a BDD, and calculating how many satisfying paths are there in a BDD. In addition to being compact, ROBDDs use far less space compared to the space required by  $n$  variables in a truth table which will require  $2^n$  lines for encoding all the possible permutations. BDDs are commonly used in digital cir-

culits and model checking and are emerging as solutions to many problems in other areas. For more details on BDDs, their data structure, algorithms, efficiency and applications, we refer the reader to [Ake78], [Lee59], [Bry86] and [Bry92].

## 3.2 Implementing Product Family Algebra using BDDs

PFA is an idempotent commutative semiring  $(S, +, 0, \cdot, 1)$ . To implement PFA using BDDs, we need to encode 0, 1, + and  $\cdot$  and hence the universe of discourse  $S$  in terms of BDDs and operations on BDDs.

We choose two models to implement PFA: the *set-model* and the *bag-model*. More models can be implemented in a similar way. The *set-model* is used in the case where duplication of features is not desired. We use the *bag-model* otherwise. Representing features and products as BDDs in the *set-model* is straightforward. However, the representation in the *bag-model* requires an extra effort. Bags can not be directly encoded in ROBDDs. This is due to the properties of ordering and the removal of duplications. We present how we implement PFA in the *set-model* followed by the *bag-model*.

### 3.2.1 The Implementation of the Set-Model

The universe of discourse  $S$  in a *set-model* is interpreted as a set of sets. We implement this as sets of BDDs (sets) to encode the product families. This means that we implement two layers of sets: the *family-level* set, and the BDD-level sets where each BDD represent a set of features.

When we adopt the *set model*, the 0 can be interpreted as the empty set at the family-level. We implement it as the empty family-level set and we write 0

in the set notation as  $\{\}$  or  $\emptyset$ . The 1 is interpreted as the singleton family-level *set* of the empty set where the empty set is represented by the empty BDD. The empty BDD is given in Figure 3.3. We write the 1 in the set notation as  $\{\{\}\}$  or  $\{\emptyset\}$ .

The  $\cdot$  in the *set-model* is implemented as the *set union* between two families. To perform the *set union* on two families  $A$  and  $B$ , we take every BDD from  $A$  *apply* the logical operation *or* to it with every BDD from  $B$ . For the  $+$  operation, it works on the family level. We implement the  $+$  as the *set union* between product families.

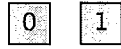


Figure 3.3: The empty BDD

We illustrate our implementation of the *set-model* PFA with the following example. Let  $A$  be a family with one product  $P_1$  composed of the features  $b$ ,  $c$  and  $d$  and let  $B$  be a family with one product  $P_2$  composed of  $f$ ,  $g$  and  $h$ . The product  $P_1$  is the set represented by the BDD of Figure 3.4 and the product in  $P_2$  is represented by the set containing the BDD of Figure 3.5.

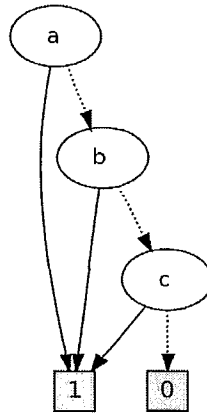
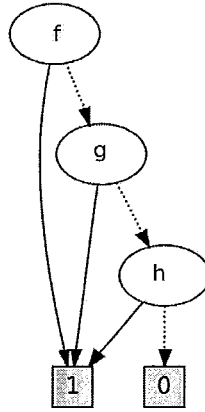
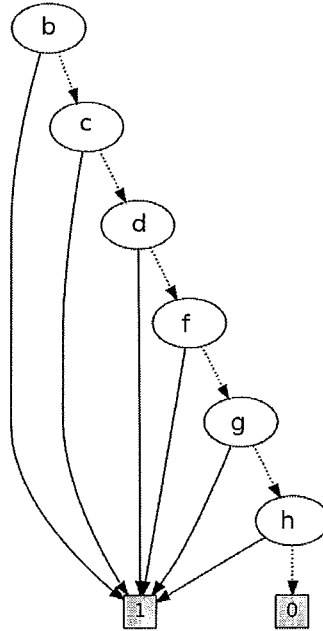


Figure 3.4: The BDD corresponding to  $P_1$

Figure 3.5: The BDD corresponding to  $P_2$ Figure 3.6: The BDD corresponding to the product in  $X$ .

If  $X = A \cdot B$ , then  $X$  is  $\{(P_1 \cup P_2)\}$ . The product of the family  $A$  and the family  $B$  is  $X$  and it has one product given in Figure 3.6. If on the other hand,  $Y = A + B$ , then  $Y$  is the *set union* on the family-level sets, i.e. the *sets* of BDDs. In this case,  $Y$  is the product family  $\{P_1, P_2\}$ .

### 3.2.2 The Implementation of the Bag-Model

If we adopt the *bag-model*, the implementation is similar to the *set-model* implementation except for handling the duplications of features. ROBDDs do not allow duplications of nodes. To handle the number of occurrences of a feature within the BDD itself, we have devised occurrence levels that encode it. We encode this number binary. For example, if we have a product with 3 features:  $x$ ,  $y$  and  $z$ , and the maximum number of occurrences of a feature is seven, then we need three binary bits to encode it. Let the product family  $W$  have one product with three  $x$  features and six  $z$  features and zero  $y$  features. Our product family contains a product  $P_w$  represented by the BDD in Figure 3.7. The BDD representing this *bag* describes the product in a way similar to that in the *set model*. However, we encode the occurrences in the levels of nodes containing  $b1$ ,  $b2$  and  $b3$ . We read in Figure 3.7 that, if  $x$  exists in this product, then we have to select  $b1$  and  $b2$ , but not  $b3$ . This is the binary code 011 representing  $b3$ ,  $b2$  and  $b1$  respectively which carries the occurrence of three for  $x$ . Similarly, for  $y$  to exist in this product, we get the binary occurrence encoding to be 000 which is 0 occurrences. For  $z$ , we get the binary occurrence of 110 which carries the number six.

In the *bag model*, we have a set of bags while in the *set model*, it is a set of sets. The difference resides in the representation of products, where they are sets in the *set model* and bags in the *bag model*. The 0 is the empty family or we can say, the empty set on the family-level. On the other hand, the 1 is the set of the empty bag (empty BDD in the bag model). The  $+$  is the *set union* on the family-level sets just as it is in the *set model*. However, the  $\cdot$  is different. In the *bag-model*, the  $\cdot$  takes into consideration the number of occurrences of the features and hence, it sums the total number of occurrences. For example, if we multiply  $U \triangleq \{(x, 3), (y, 0), (z, 6)\}$  with  $V$  where  $V \triangleq \{(y, 1), (z, 1)\}$ . The

result is the product family  $Z = \{(x, 3), (y, 1), (z, 7)\}$ . This product family contains one product  $P_z$  represented by the BDD given in Figure 3.8. We read the number of occurrences of  $x$ ,  $y$ , and  $z$  in  $P_z$  in a similar way as we did for  $P_w$ .

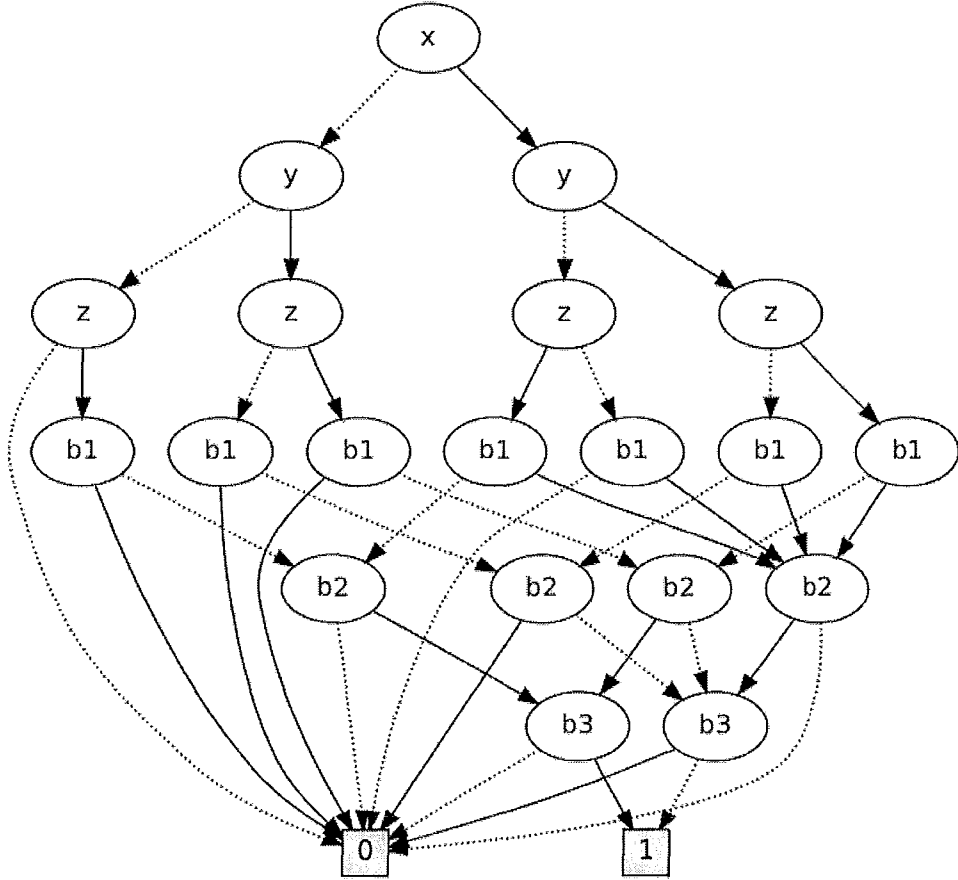
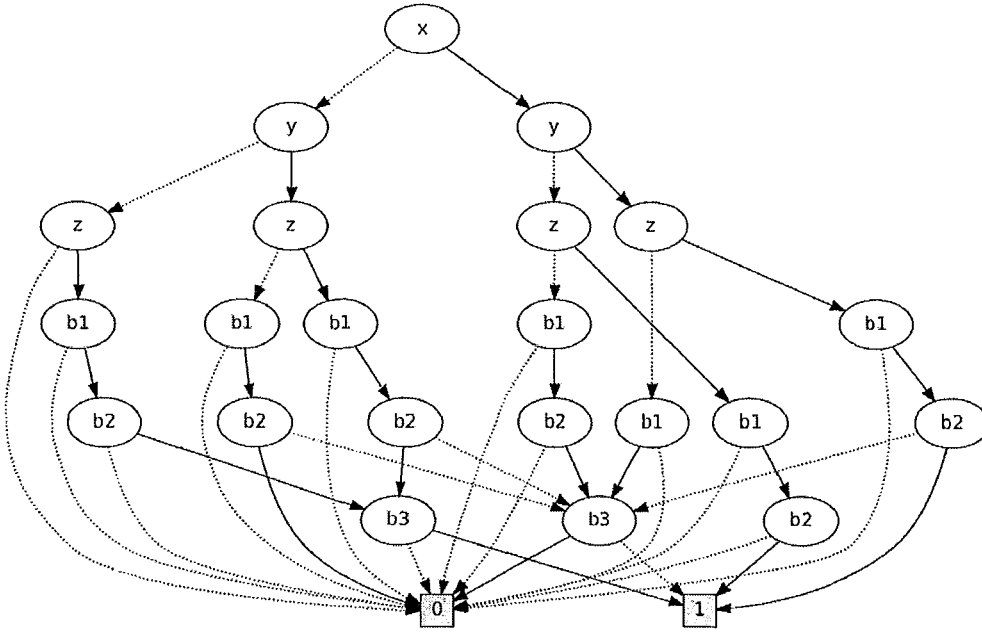


Figure 3.7: The BDD corresponding to  $P_w$  in  $W$

We now handle a more delicate example; our familiar bicycle example. For brevity, we use the set notation. We first illustrate how PFA works in this example in the *set model* and then in the *bag model*. The basic features are in  $BF$  where  $BF = \{\text{handle\_bar, handle\_grip, shock\_absorber, fork, hub\_gears, derailleur\_gears, front\_brakes, rear\_brakes, top\_tube, down\_tube, seat\_tube, seat\_stay, chain\_stay, saddle, saddle\_post, tire, spokes, rim, hub, chain, chain\_rings, pedal, crank}\}$ .

Figure 3.8: The BDD corresponding to  $P_z$  in  $Z$ 

For the specification of the bicycle, we write in PFA:

```

front_set = handle_bar . handle_grip . fork . (shock_absorber + 1)
gears_set = hub_gears + derailleur_gears
brakes_set = front_brakes . rear_brakes
frame = top_tube . down_tube . seat_tube . seat_stay . chain_stay
saddle_set = saddle . (saddle_post + 1)
wheels_set = tire . spokes . rim . hub
chain_set = chain . chain_rings
pedal_set = pedal . pedal . crank . crank
bicycle = front_set . gears_set . brakes_set . frame .
          saddle_set . wheels_set . wheels_set . chain_set . pedal_set

```

We now consider this specification in the *set model*. For the  $\cdot$  in the *pedal\_set*, we get  $pedal\_set = \{\{pedal, crank\}\}$ ; a family of one product composed of one *pedal* and one *crank*. We notice that multiple occurrence of *pedal* and *crank* is neglected. This is because the *set model* does not allow the duplication of features. For the  $+$ , we have the *gears\_set* and we get  $gears\_set = \{\{derailleur\_gears\}, \{hub\_gears\}\}$ ; a family of two products representing two choices of gears. The *front\_set* family is expressed in terms of  $\cdot$ ,  $+$  and  $1$ . For this specification, we get  $front\_set = \{\{handle\_bar, handle\_grip, fork\}, \{handle\_bar, handle\_grip, fork, shock\_absorber\}\}$ . The *front\_set* is a family of two products to choose from, one with the *shock\_absorber* and one without. We express that a feature is optional using the  $+$  and the  $1$ . For example, in the *front\_set* expression we have the  $(shock\_absorber + 1)$  composed with the other features using the  $\cdot$  operation.

If we consider the *bag model*, we notice some differences in the results. For the  $\cdot$  in the *pedal\_set* we get  $pedal\_set = \{(pedal, 2), (crank, 2)\}$  with two occurrences of the *pedal* and two occurrences of *crank* as the *bag model* allows the duplications of features. For the  $+$ , it behaves similarly as in the *set model* for the *gears\_set* and the other specifications.

Using PFA, we can calculate the number of potential products of the bicycle family. In both models, the number of potential products is eight. The difference happens at the lower level and it does not affect the number of products in the family level. The family *bicycle* in the *bag model* has eight products that we list below:

```
{
  {
    (handle_bar,1) , (handle_grip,1) , (fork,1) , (derailleur_gears,1) ,
    (front_brakes,1) , (rear_brakes,1) , (top_tube,1) , (down_tube,1) ,
```



```

(seat_tube,1) , (seat_stay,1) , (chain_stay,1) , (saddle,1) ,
(tire,2) , (spokes,2) , (rim,2) , (hub,2) , (chain,1) ,
(chain_rings,1) , (pedal,2) , (crank,2)
}

,

{
(handle_bar,1) , (handle_grip,1) , (fork,1) , (derailleur_gears,1) ,
(front_brakes,1) , (rear_brakes,1) , (top_tube,1) , (down_tube,1) ,
(seat_tube,1) , (seat_stay,1) , (chain_stay,1) , (saddle,1) ,
(saddle_post,1) , (tire,2) , (spokes,2) , (rim,2) , (hub,2) ,
(chain,1) , (chain_rings,1) , (pedal,2) , (crank,2)
}

,

{
(handle_bar,1) , (handle_grip,1) , (fork,1) , (hub_gears,1) ,
(front_brakes,1) , (rear_brakes,1) , (top_tube,1) , (down_tube,1) ,
(seat_tube,1) , (seat_stay,1) , (chain_stay,1) , (saddle,1) ,
(tire,2) , (spokes,2) , (rim,2) , (hub,2) , (chain,1) ,
(chain_rings,1) , (pedal,2) , (crank,2)
}

,

{
(handle_bar,1) , (handle_grip,1) , (fork,1) , (hub_gears,1) ,
(front_brakes,1) , (rear_brakes,1) , (top_tube,1) , (down_tube,1) ,
(seat_tube,1) , (seat_stay,1) , (chain_stay,1) , (saddle,1) ,
(saddle_post,1) , (tire,2) , (spokes,2) , (rim,2) , (hub,2) ,
(chain,1) , (chain_rings,1) , (pedal,2) , (crank,2)
}

```

```

}

,

{
(handle_bar,1) , (handle_grip,1) , (shock_absorber,1) , (fork,1) ,
(derailleur_gears,1) , (front_brakes,1) , (rear_brakes,1) ,
(top_tube,1) , (down_tube,1) , (seat_tube,1) , (seat_stay,1) ,
(chain_stay,1) , (saddle,1) , (tire,2) , (spokes,2) , (rim,2) , (hub,2) ,
(chain,1) , (chain_rings,1) , (pedal,2) , (crank,2)
}

,

{
(handle_bar,1) , (handle_grip,1) , (shock_absorber,1) , (fork,1) ,
(derailleur_gears,1) , (front_brakes,1) , (rear_brakes,1) , (top_tube,1)
(down_tube,1) , (seat_tube,1) , (seat_stay,1) , (chain_stay,1) ,
(saddle,1) , (saddle_post,1) , (tire,2) , (spokes,2) , (rim,2) , (hub,2)
(chain,1) , (chain_rings,1) , (pedal,2) , (crank,2)
}

,

{
(handle_bar,1) , (handle_grip,1) , (shock_absorber,1) , (fork,1) ,
(hub_gears,1) , (front_brakes,1) , (rear_brakes,1) , (top_tube,1) ,
(down_tube,1) , (seat_tube,1) , (seat_stay,1) , (chain_stay,1) ,
(saddle,1) , (tire,2) , (spokes,2) , (rim,2) , (hub,2) ,
(chain,1) , (chain_rings,1) , (pedal,2) , (crank,2)
}

,

{

```

```

(handle_bar,1) , (handle_grip,1) , (shock_absorber,1) , (fork,1) ,
(hub_gears,1) , (front_brakes,1) , (rear_brakes,1) , (top_tube,1) ,
(down_tube,1) , (seat_tube,1) , (seat_stay,1) , (chain_stay,1) ,
(saddle,1) , (saddle_post,1) , (tire,2) , (spokes,2) , (rim,2) ,
(hub,2) , (chain,1) , (chain_rings,1) , (pedal,2) , (crank,2)
}
}

```

We can also extract the commonality features in the *bicycle* family. All the above products share the following features.

```

{
(handle_bar,1) , (handle_grip,1) , (fork,1) , (front_brakes,1) ,
(rear_brakes,1) , (top_tube,1) , (down_tube,1) , (seat_tube,1) ,
(seat_stay,1) , (chain_stay,1) , (saddle,1) , (tire,2) , (spokes,2) ,
(rim,2) , (hub,2) , (chain,1) , (chain_rings,1) , (pedal,2) , (crank,2)
}

```

To the bicycle, we add three more features and we define new families. We add a basic feature *mirror*, and we define two bicycles: *hub\_bicycle* and *mirror\_bicycle*. The *hub\_bicycle* is similar to *bicycle* but we explicitly specify that it can only have the *hub\_gears*. The *mirror\_bicycle* is a *hub\_bicycle* with additional two mirrors - we are still in the *bag model*. We add one more basic feature *mirror* to the the set  $BF$  and we append this specification to the above.

```

hub_bicycle = front_set . hub_gears . brakes_set . frame . saddle_set .
              wheels_set . wheels_set . chain_set . pedal_set
mirror_bicycle = hub_bicycle . mirror . mirror

```

We can check whether *hub\_bicycle* is a subfamily of the *bicycle*. We notice that *hub\_bicycle* is a family of four products.

```

{
  { handle_bar, handle_grip, fork, hub_gears, front_brakes, rear_brakes,
    top_tube, down_tube, seat_tube, seat_stay, chain_stay, saddle, tire,
    spokes, rim, hub, chain, chain_rings, pedal, crank
  }
  ,
  { handle_bar, handle_grip, fork, hub_gears, front_brakes, rear_brakes,
    top_tube, down_tube, seat_tube, seat_stay, chain_stay, saddle,
    saddle_post, tire, spokes, rim, hub, chain, chain_rings, pedal, crank
  }
  ,
  { handle_bar, handle_grip, shock_absorber, fork, hub_gears, front_brakes,
    rear_brakes, top_tube, down_tube, seat_tube, seat_stay, chain_stay, saddle,
    tire, spokes, rim, hub, chain, chain_rings, pedal, crank
  }
  ,
  { handle_bar, handle_grip, shock_absorber, fork, hub_gears, front_brakes,
    rear_brakes, top_tube, down_tube, seat_tube, seat_stay, chain_stay, saddle,
    saddle_post, tire, spokes, rim, hub, chain, chain_rings, pedal, crank
  }
}

```

Comparing the products in the *hub\_bicycle* family to the products in the *bicycle* family, we see the first is a subfamily of the second. The *mirror\_bicycle* family is given as follows:

```
{
  { handle_bar, handle_grip, fork, hub_gears, front_brakes, rear_brakes,
    top_tube, down_tube, seat_tube, seat_stay, chain_stay, saddle, tire,
    spokes, rim, hub, chain, chain_rings, pedal, crank, mirror
  }
  ,
  { handle_bar, handle_grip, fork, hub_gears, front_brakes, rear_brakes,
    top_tube, down_tube, seat_tube, seat_stay, chain_stay, saddle,
    saddle_post, tire, spokes, rim, hub, chain, chain_rings, pedal,
    crank, mirror
  }
  ,
  { handle_bar, handle_grip, shock_absorber, fork, hub_gears, front_brakes,
    rear_brakes, top_tube, down_tube, seat_tube, seat_stay, chain_stay,
    saddle, tire, spokes, rim, hub, chain, chain_rings, pedal, crank,
    mirror
  }
  ,
  { handle_bar, handle_grip, shock_absorber, fork, hub_gears, front_brakes,
    rear_brakes, top_tube, down_tube, seat_tube, seat_stay, chain_stay,
    saddle, saddle_post, tire, spokes, rim, hub, chain, chain_rings, pedal,
    crank, mirror
  }
}
```

```

    }
}

```

Again, we have four products in the *mirror\_bicycle* family. We have just made the bicycles more fancy with the mirrors. The *mirror\_bicycle* is not a subfamily of the *bicycle* and *hub\_bicycle* families. However, the *mirror\_bicycle* refines the *hub\_bicycle* and refines the *bicycle* families. This is because *every* element in the *mirror\_bicycle* has the same features or more of *some* of the products in *hub\_bicycle* and the *bicycle* families. The *hub\_bicycle* family also refines the *bicycle*. This is because *all* the elements of the first has the same features or more of *some* elements in the second.

As we have seen, with the mathematics inherited from the idempotent commutative semiring and the mathematics of sets and bags, PFA can provide extensive capabilities for specification, calculation, analysis and inference. We have implemented the above basic functionality illustrated in the bicycle example in our tool *Jory*.

### 3.3 Conclusion

To handle large and critical software families and product lines, we have chosen to implement the technique PFA based on BDDs. The PFA is a rigorous mathematical technique which extends the benefits of the graphical FM techniques with formalism needed for critical systems. With BDDs, we bring efficiency to handling large systems. We use BDDs to implement two models for PFA, the *set model* and the *bag model*. In the *set model*, a family is represented by a set of sets and in the *bag model*, it is represented by set of bags. The upper level set is the family-level set and the lower level sets are represented by BDDs as an

aggregation of features. We have defined the operations of PFA as operations on BDDs. More over, based on these operations and the mathematics of sets and bags, we get a rich functionality.





# Chapter 4

## System Design

In this chapter, we lay out the system design. We first give the architectural design then follow with the detailed design. We also present the major design decisions, the capabilities, and some possible extensions to the proposed design. The proposed design is for the system as we envision it at this point in time. However, a layer of its implementation is intended to be the subject of future work.

### 4.1 Architectural Design

We have designed our system in a layered architecture. The layered architecture helps us maintain, modify, extend and replace the layers independently. Starting from the top layer in Figure 4.1, we have the *User Interface Layer*, the *Translation Layer*, the *Term Evaluation Layer*, the *Concrete Models Layer*, and the *BDD Layer*. The shadowed boxes are subjects of future work.

The *User Interface Layer* enables the user to input the specification either in PFA or as a graphical feature model. The user inputs the specification written in

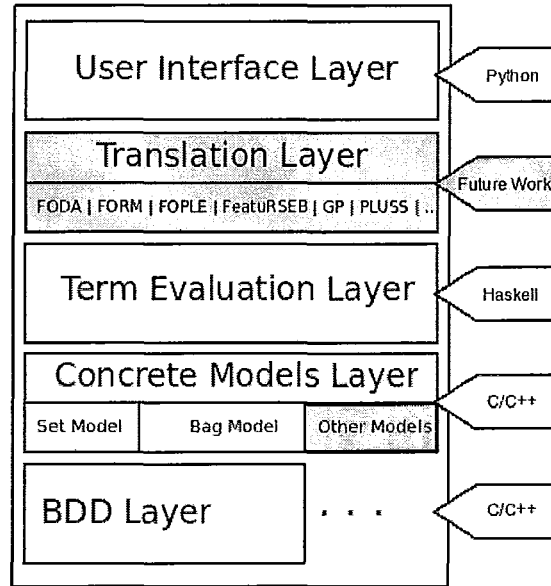


Figure 4.1: Architectural Design

the language of PFA in terms of features, products, and families. The interface also enables the user to configure the system, set the environment and set the preferences such as selecting one of the models of PFA.

The *Translation Layer* provides a means to translate from one FM techniques to another such as from FODA to PLUS5. It also provides a way to translate from PFA specification to a feature model in a technique like FeatuRSEB and vice versa. This layer establishes the bridge of communication between PFA and graphical FM techniques. It also converts from one graphical feature model to another.

The *Term Evaluation Layer* takes the specification in PFA whether it is supplied in PFA language or is generated by the translation layer from a graphical model. This layer analyses and evaluates the terms in a specification

and generates a registry of the features, products, and families and it associates them with their extracted classifications and description data.

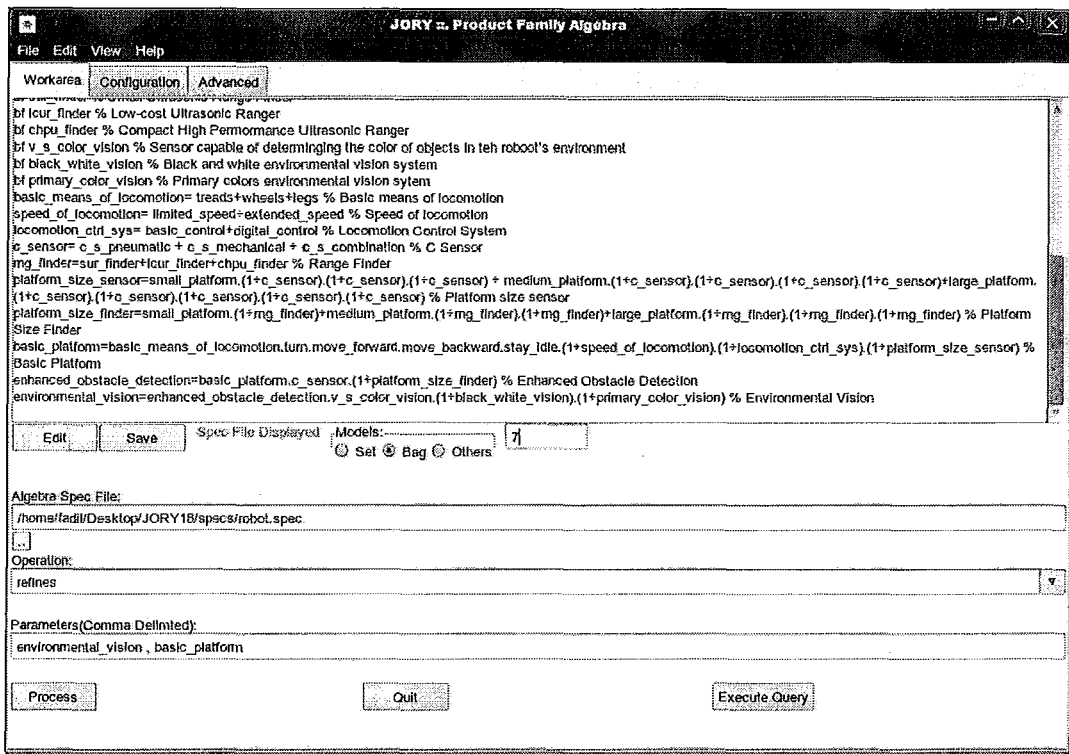
The *Concrete Models Layer* handles the implemented models of PFA such as the *set model* and the *bag model* or any other model that might be useful. The registry from the above layer is used to generate the code in the user selected concrete model in terms of BDD code.

The *BDD Layer* is the lowest layer where the BDD code - generated in the above layer in the selected concrete model - is executed. This is where the specification, the queries, and transactions take place and the results are produced.

The layers communicate top down and down up. The specification is either supplied or extracted from a feature model. It is then evaluated and a registry is created. The concrete models are instantiated and the BDD code is generated. The code executes and the results are pushed up towards the user interface. This layered architecture takes into consideration the need for efficiency and scalability. The layers can be plugged in, removed, and replaced whenever needed.

## 4.2 Detailed Design

In this section we give the detailed design of the tool. We take the order of the layers from top to down and give an overview of every layer in terms of their modules.

Figure 4.2: A Snapshot of the Interface of *Jory*

### 4.2.1 The Interface Layer

The interface layer is written in Python. The interface has the main tab which is the work area. The work area has a text editor for the input of the specification that is used as well for showing the results. This tab also has a text control for the name of the specification file, radio buttons for selecting the *set model* or the *bag model* and setting the maximum number of occurrences a feature can have if the *bag model* is selected. This tab also has a list box of the main functions that can be applied on the product families such as *size*, *list products*, *is sub-family* and *refines*. There is also a field to provide the parameters for the above operations. Figure 4.2 is a snapshot of the interface of *Jory*.

The interface also has another two tabs, both for the configuration of the

system environment. This enables the user to specify the default specification file, the directories, the utility files and the result files.

### 4.2.2 The Translation Layer

The translation layer is for translating from one graphical feature modelling technique to another. It is also used to translate from a graphical feature model in a technique to a PFA specification and vice versa. This is designed to perform the translation using syntax-directed translation [Pet71]. We propose that the graph files are supplied in *Graphvis Dot* [LC10] format. We then map syntax to syntax translation rules. This layer is currently not implemented.

### 4.2.3 The Term Evaluation Layer

This layer is written in Haskell. We put the modules into two groups. We call the first group the *Specification Analysis Group* and it contains modules to process the specification files and generate the register of features, products, and families. We call the other the *Evaluation Group* and it contains the modules to evaluate and assign the expressions in the register. The design of this layer is given in Figure 4.3.

#### 4.2.3.1 The Specification Analysis Group

- **Module:** *SpecAnalysis*.

**Secret:** It is an algorithm that reads a specification file with the extension '.spec'.

**Service:** It calls *LineSpecAnalysis*, *TermAnalysis*, *Infix2Prefix* and *Variable\_Constrain\_Registration* module. The major task is to take a specification file and produce a register file with '.reg' extension. The reg-

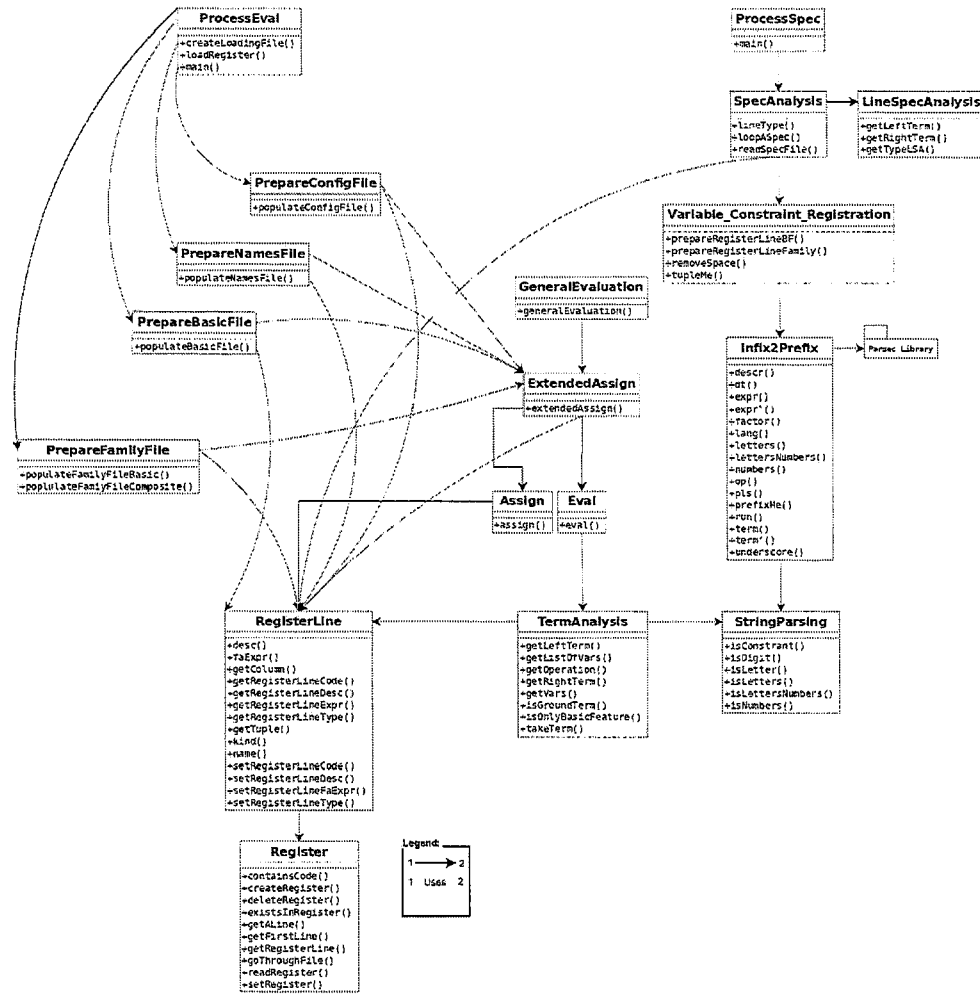


Figure 4.3: The Term Evaluation Layer

ister file contains all the basic features, products and families.

- **Module:** *LineSpecAnalysis*.

**Secret:** It is an algorithm that handles a specification line.

**Service:** This module handles a specification line from the specification file. It returns the specification line type if it is a specification of a feature, a product or a family. It also returns the right and the left terms of the line.

- **Module:** *TermAnalysis*.

**Secret:** It is an algorithm that handles terms.

**Service:** This module analyses the terms returned from the *LineSpecAnalysis* module. It breaks the terms into ground terms (i.e. terms without variables), variables, and operations. The returned values are required for creating the register.

- **Module:** *Register*.

**Secret:** It is a data structure.

**Service:** This module has the operations on registers: creating, deleting, searching, looping through, reading and setting a register and returning register lines.

- **Module:** *RegisterLine*.

**Secret:** It is a data structure.

**Service:** It contains the operations on register lines. It gets and sets the line code, description, expression, and the type. The code stands for the name for the feature, product or the family expressed in the register line. The expression is in PFA expression. The description is the user free text to describe the code and the type specifies if the line is for a feature, a

family or a constraint.

- **Module:** *Variable\_Constraint\_Registration*.

**Secret:** It is a data structure (a tuple).

**Service:** This module handles the variables in a register line and prepares a line for a basic feature, a family, or a constraint.

- **Module:** *Infix2Prefix*.

**Secret:** It is an algorithm that converts infix to prefix notation.

**Service:** This module converts a PFA expression from the user supplied infix notation to a prefix notation.

#### 4.2.3.2 The Evaluation Group

- **Module:** *Eval*.

**Secret:** It is an algorithm.

**Service:** This module evaluates the ground terms which are 0 and 1.

- **Module:** *ExtendedAssign*.

**Secret:** This is an algorithm to evaluate expressions.

**Service:** This module evaluate the expressions composed of variables and ground terms.

- **Module:** *Assign*.

**Secret:** It is an algorithm to evaluate variables of basic features.

**Service:** This module evaluate variables which are basic features.

- **Module:** *GeneralEvaluation*.

**Secret:** It is an algorithm to manage the evaluation modules.

**Service:** This module controls the evaluation and assignment modules.



There are also other modules in this layer to generate the configuration files, the registry files for basic features and for families. There is also a module to control the overall specification analysis process and another to control the evaluation process.

#### 4.2.4 The Concrete Models Layer

This layer contains a module for sets, a module for bags, and a module to convert from sets to bags and vice versa. More modules can be written to accommodate any future desired concrete model for PFA. The modules of this layer are given in Figure 4.4

- **Module:** *Set*.

**Secret:** It is a data structure.

**Service:** This module implements BDDs as sets. It also provides set operations such as `setComplement`, `setEnumerate`, `insert`, `setIntersection`, and `setUnion`.

- **Module:** *Bag*.

**Secret:** It is a data structure.

**Service:** This module implements BDDs as bags. It also provides bag operations such as `bagComplement`, `bagEnumerate`, `insert`, `bagIntersection`, `bagUnion`, `getOccurrence`, and `setOccurrence`.

- **Module:** *Set2Bag*.

**Secret:** It is an algorithm to convert from set to bag and vice versa.

**Service:** This module converts sets to bags and bags to sets.

### 4.2.5 The BDD Layer

This layer is implemented using two libraries written in C/C++. The Buddy library by Jørn Lind-Nielsen [LN10] for BDDs and the STL library [SGI10] for sets. We use the first one to implement PFA in terms of BDDs with a module for sets and another for bags. The other one is used to implement the family-level sets. These are the containers that represent the families of BDDs. For the design of this layer, see Figure 4.4.

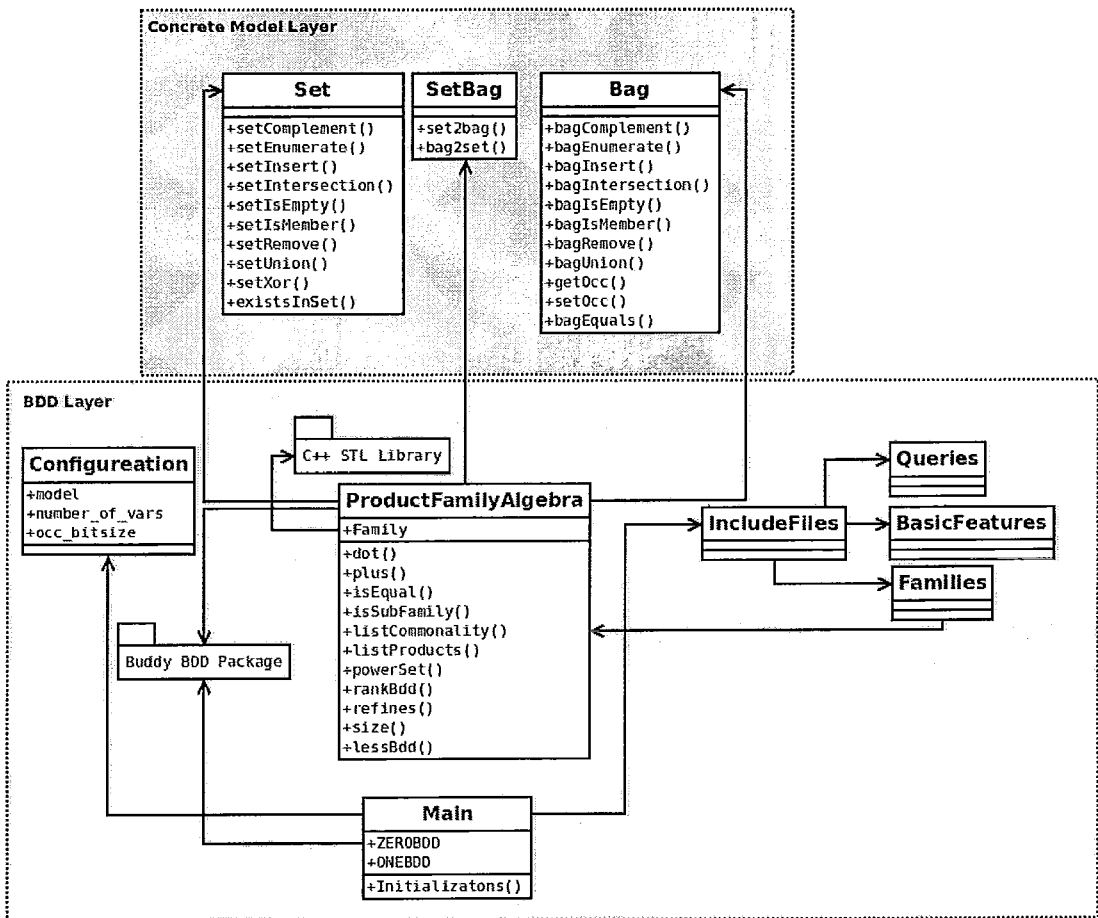


Figure 4.4: Concrete Models' Layer and the BDD Layer

- **Module:** *ProductFamilyAlgebra*.

**Secret:** It is a data structure.

**Service:** This is the core module in this layer. We implement the the data type family using the C++ STL library and we implement the operations  $\cdot$ ,  $+$ , and the operations that the user frequently need to perform on families. Due to the importance of this module, we dedicate Section 4.2.5.1 for discussing its design.

- **Module:** *Queries*.

**Secret:** It is a data structure.

**Service:** This file contains the queries required by the user to be performed on the user selected families.

- **Module:** *BasicFeatures*.

**Secret:** It is a data structure.

**Service:** This file contains the list of basic features. It is used to generate BDD code of the specification basic features.

- **Module:** *Families*.

**Secret:** It is a data structure.

**Service:** This file contains the list of families. The file is used generate the BDD code for the specified products and families.

#### 4.2.5.1 ProductFamilyAlgebra Module

This module has the following access programs.

- **Access Program:** *DOT*.

**Service:** Depending on the model selected; set or bag, the function takes two families and applies the  $\cdot$  operation on them. It multiplies every BDD in the first family with every BDD in the second. If the model is the *set*

*model*, it calls the *set union* from the set module, otherwise, it calls the *bag union* from the bag module in the concrete layer.

- **Access Program:** *PLUS*.

**Service:** This function applies the  $+$  on two families. The is the the *set union* operates on the family-level sets. We implement this *set union* using the *STL Set* library.

- **Access Program:** *isEqual*.

**Service:** This function returns true if the compared two families are equal.

- **Access Program:** *isSubFamily*.

**Service:** This function returns true if the first family is a subfamily of the second. It implements the ordering relation  $\leq$  that we presented in section 2.2.1.

- **Access Program:** *refines*.

**Service:** This function returns true if the first family refines the second. It implements the refinement relation  $\sqsubseteq$  that we presented in section 2.2.1.

- **Access Program:** *listProducts*.

**Service:** This function returns a list of products in a family.

- **Access Program:** *listCommonality*.

**Service:** This function returns a list of features that are common in all the products in a family. The list is obtained by getting the common factors of the given family features.

- **Access Program:** *powerSet*.

**Service:** This function takes a family and generates the power set based on the features in the given family.

- **Access Program:** *rankBDD*.

**Service:** This function ranks the given BDD. It takes a BDD and it returns its rank as an array to represent the existence of the variables or their number of occurrences for sets or bags respectively. Unique BDDs have unique ranks.

- **Access Program:** *size*.

**Service:** This function takes a family and returns the size which is the number of products it contains.

- **Access Program:** *lessBbdd*.

**Service:** This is a comparator function. It is used with the *rankBDD* function to order the BDDs in the families (family-level sets).

Adopting the principle of the *separation of concerns* in our design and the implementation, makes the code simple unit that are easy to maintain. Every unit of code is once service and it becomes easy to manage, modify, validate and test. Here are some code snippets of the PFA module.

The *DOT* function code:

```
FAMILY DOT(FAMILY const &f1, FAMILY const &f2)
{
    bdd bdd_result;
    FAMILY family_result;
    FAMILY::iterator iF1,iF2;
    if (f1.empty() | f2.empty() ) return family_result;
    else
    {
        for (iF1 = f1.begin(); iF1 != f1.end(); iF1++ )
            for (iF2 = f2.begin(); iF2 != f2.end(); iF2++ )
```

```

        {
            if (model ==0) family_result.insert(setUnion(*iF1,*iF2));
            else          family_result.insert(bagUnion(*iF1,*iF2));
        }

    return family_result;
}

return family_result;
}

```

The *PLUS* function:

```

FAMILY PLUS(FAMILY const &f1, FAMILY const &f2)
{
    FAMILY::iterator iF1, iF2;
    FAMILY family_result;
    set_union(f1.begin(), f1.end(), f2.begin(), f2.end(),
              inserter(family_result, family_result.end()), less_bdd());
    return family_result;
}

```

The *listProducts* function:

```

void listProducts(FAMILY prod)
{
    FAMILY::iterator iF1;
    cout << "[ " << endl;
    for (iF1 = prod.begin(); iF1 != prod.end(); iF1++ )
    {
        if (model==0) setEnumerate(*iF1);
        else          bagEnumerate(*iF1);
    }
}

```

```

    cout << "]" << endl;
}

```

The *powerSet* function:

```

FAMILY powerSet(FAMILY f)
{
    FAMILY head, tail, emptySet;
    FAMILY::iterator iCurr, iF1;
    tail = f;
    emptySet.insert(bddfalse);
    if (f.empty()==true) return f;
    else
    {
        iCurr = tail.begin();
        head.insert(*iCurr);
        tail.erase(iCurr);
    }
    return PLUS(emptySet, PLUS(head, PLUS(powerSet(tail), DOT(head, powerSet(tail)))));
}

```

The *isSubFamily* function:

```

bool isSubFamily(FAMILY f1, FAMILY f2)
{
    bool found = false;
    if (isEqualInternal(PLUS(f1,f2),f2)) found = true;
    if (found == true)          cout << "is SubFamily? Yes." << endl;
    else                        cout << "is SubFamily? No." << endl;
}

```

## 4.3 Major Decisions, Capabilities, and Future Extensions

One of our aims is to make the tool platform-independent. We also consider that the tool handles large and critical specifications efficiently and accurately. This contributed to the motivation of selecting *Python* for the interface, *Haskell* for the evaluation layer, and C/C++ for the concrete and the BDD layers. We have also used efficient, yet freely available libraries: Buddy for BDDs and C++ STL for the family-level sets, and Parsec [Utr10] for parsing.

We choose the Buddy library as it is freely available and is one of libraries for BDDs that is efficient and provides the core BDD algorithms. According to [RZS09], the Buddy library shown to be one of the most efficient libraries compared with the other available libraries. Buddy handles 50,000 nodes per one megabyte of memory and if there is no memory limits, it can handle up to  $2^{32}$  nodes efficiently and a decision is made in polynomial time with respect to the number of variables used in a BDD [Gee03].

We choose the C++ STL (Standard Template Library); a free library published by SGI. It provides a set of classes, containers, iterators and algorithms that work with user-defined data structures. We use the STL *Set* class to implement the family-level sets that contain BDDs. We have defined an ordering relation to maximize the efficiency of building, comparing, and handling the *sets of sets* or the *sets for bags*. The ordering relation is based on the *rankBDD* function that we ranks BDDs uniquely and it is introduced in Section 4.2.5.1.

Haskell is a powerful pure functional programming language which provides



rich functionality of string manipulations and processing. We chose Haskell to write the term evaluation layer for the analysis and the evaluation of the specification files. We also used the industry-strength Parsec library for implementing the conversion of infix to prefix notation. This is needed for converting from the infix notation of the user specification to the prefix notation that is used for writing the BDD code.

Python is a multi-paradigm language (functional, object-oriented and imperative) providing a rich library for graphical user interfaces. We used it to build the user interface layer and the communication channel between the layers. In the future, it can be used to build another editor for graph feature models to be based on the *Graphviz Dot* language.

We have designed the tool for change; to handle future capabilities and extensions. There are many plausible extensions that can be done easily with the current design and implementation. For example, we can facilitate the PFA language with macros such as writing  $\text{opt}[x]$  in the specification, instead of writing  $(1 + x)$  for a feature  $x$ . The macros can also include constraints like  $x$  *requires*  $y$  in  $f$  or  $x$  *excludes*  $y$  in  $f$  where  $x$  and  $y$  are features and  $f$  is a family. The conversion from PFA specifications to the graphical FM techniques and from one FM technique to another is also a useful extension. We propose that this is done via adopting the *Graphviz Dot* language to input graphs. The conversion can be done via the syntax-based translation. The *Dot* files for the graphical feature models can also be translated in the same way to PFA specification.

The tool incorporating the conversion from and to PFA specifications can be used to customise the visualization of the feature models instantaneously with the facility of searching, highlighting, zooming in and zooming out and

sub-graphing feature models. This also enables the tool to use the visual editor to manage the merging or splitting feature models.

# Chapter 5

## Testing and Validation

This chapter is a report on the testing and the validation of *Jory*. We first address the testing techniques applied. We give an overview of each technique, its objective, and a summary of the results obtained. We follow then with validation and different walk-through validation examples and summarize the results.

### 5.1 Testing Techniques Adopted

We have adopted four testing techniques to test our tool *Jory*. We adopted *unit testing*, *integration testing*, *parallel testing*, and *acceptance testing*.

#### 5.1.1 Unit Testing

In the unit testing, we tested every function of the system. The objective is to verify that each function supplied with the input behaved as desired and produced correct results. We have various input ranges including extreme limits. Each unit was tested individually independently and the discovered errors were corrected. We have started with the primitive functions, those that do not require other functions that we wrote. Gradually, we tested those functions that

require other functions that are already tested. For example, in the lower layers of the systems, we tested the functions in the *set* module and the *bag* module then we tested the functions that require them in the *Product Family Algebra* module. We followed this approach in all the layers starting from primitive independent functions to those which call other functions in each module and in each layer.

### 5.1.2 Integration Testing

The integration testing took place after the unit testing. The objective is to assess whether the modules collaborate together as required. We have integrated the layers gradually from the bottom up. We started testing the *BDD Layer* integrated with the *Concrete Models Layer*. We have tested the two layers together and once they worked well, we integrated them with the *Term Evaluation Layer*. We continued in this way until all the layers integrated and tested to work as desired. We have conducted tests where we verify the communication between layers top down and down up is successful. We have traced the communication channels between the layers. We have tested the same specification files fed to the top layer at each integration stage, and monitored the behaviour of the system. The specification at each stage was having a different format as it is a BDD code at the lower levels, a registry at the *Concrete Layer* and a specification file in PFA language at the interface level. At every integration stage, we tested the system and moved to the next stage as the current one worked correctly until overall integration was completed.

### 5.1.3 Parallel Testing

We have also conducted parallel testing. We have tested *Jory* with the prototype tool written previously in Haskell in [HKM06]. The objective was to verify whether *Jory* produces correct results and to compare the performance. The previous tool was built using Haskell and the families were implemented as lists of lists. We have made redundant executions of the same examples on both tools and we had the same results. For example, we have executed the *Employee Self Service* example on both. The number of products in the family *employee\_self\_service* given by both tools was 432 products. We have compared the basic functions available in both tools such as the size, the listing of products, listing the commonality features in a family, checking refinement and checking whether a family is a subfamily of another. We find that *Jory* however, preceded the prototype tool in the speed of execution. This is clearly observable when we handle larger specification files such as the *Robot* example found in Section 5 of [HKM06]. It involves 23 basic features. In the *set model*, *Jory* takes about 15 seconds to show the size of the *environmental\_vision* while it takes about six hours for the prototype tool to get the same result. For the same family in the *bag* model with the maximum number of occurrences set to seven, it takes about an hour to show the number of products while it takes about 48 hours for the prototype to do so. *Jory* showed faster execution. We present specification of the *Robot* example in Section 5.2.

### 5.1.4 Acceptance Testing

The acceptance testing was carried out to verify that the system meets the user requirements. We thank Qinglei Zhang who took the time and tested the system

thoroughly using various examples. She used examples that vary in size and nature. The examples were tested in the *set* and the *bag* models. The examples assessed the basic functionalities of the tool and the provided services via the interface. We have made modifications and corrections to the system as some bugs were discovered.

## 5.2 Validation Examples

We have validated the system through examples that differ in the number of basic features and in their composition. In this section, we present three examples that we have mentioned above and show how the tool handles feature modelling. The examples are the *Employee Self Service* which is well-suited for the *set model* and the *Robot* which is well-suited for the *bag model*.

### 5.2.1 The Employee Self Service

This is the specification file for the *Employee Self Service* example adopted from [HKM06]. This is a specification of human resource employee self services system. A basic feature in the specification is preceded with the keyword "bf" then its description follows. We use "%" to separate the label of a feature or a product family from its description. We then give the specification of products and families based on the basic features.

#### 5.2.1.1 Specification

```
bf personal_info % personal info
bf personal_info_flexibility % personal info flexibility
bf basic_personal_tasks % basic personal tasks
bf update_personal_info % update personal info
```



```

% ess payroll and benifits tasks

employee_self_service= internet_intranet_enable_ess

    . ess_basic_personal_tasks
    . ess_time_and_attendance_tasks
    . ( 1 + expense_tasks )
    . ess_payroll_and_benefits_tasks

% employee self service whole system

```

Looking at *employee\_self\_service*, we see that it is composed of *internet\_intranet\_enable\_ess*, *ess\_basic\_personal\_tasks*, *ess\_time\_and\_attendance\_tasks*, an optional *expense\_tasks*, and *ess\_payroll\_and\_benefits\_tasks*.

#### 5.2.1.2 Results

The specification is written in a file with ".spec" extension. We load this file into the tool, we choose the *set model* and we execute the *size* operation on the family *ess\_time\_and\_attendance\_tasks*. We get the size 36. The size of the family *ess\_payroll\_and\_benefits\_tasks* is 6 products and the size of *employee\_self\_service* is 432 products. We use the operation *listProducts* of the *ess\_payroll\_and\_benefits\_tasks*, we get:

```

[
{ }
{ multi_benefit_programs }
{ benefit_display }
{ payroll_administration }
{ payroll_administration, multi_benefit_programs }
{ payroll_administration, benefit_display }
]

```



For example, we also can show the commonality in a family, *employee\_self\_service*. The tool returns the following.

```
[
{ personal_info, personal_info_flexibility, basic_personal_tasks,
  update_personal_info, employment_history, hr_policies
}
]
```

We also can check whether two families are equal, one is a subfamily of another, or whether one refines another. For the refinement, we check if *employee\_self\_service* refines *ess\_payroll\_and\_benefits\_tasks* and it shows that it does.

For this case, we do not have any feature duplication. This makes the results in the *bag model* identical to those that we got in the *set model* except for the way they are displayed. The *bag model* output includes the cardinality of 1 for every feature that exists in a product in a family. We switch to the *bag model* and we set the maximum number of occurrences for a feature to three. We display the result for *listProducts* of the family *ess\_payroll\_and\_benefits\_tasks* and it is:

```
[
{

}
{
  (multi_benefit_programs,1)
}
{
```

```

    (benefit_display,1)
  }
  {
    (payroll_administration,1)
  }
  {
    (payroll_administration,1) , (multi_benefit_programs,1)
  }
  {
    (payroll_administration,1) , (benefit_display,1)
  }
]

```

We execute the other operations on the families in the *bag model* and get similar results to those we get in the *set model*.

## 5.2.2 The Robot Example

The specification file for the *Robot* example which is adopted from [HKM06], is given as follows. This specification has multiple occurrences and hence we need the bag model to handle it. We follow the same manner in starting with defining the basic features of a robot and we follow with products and families.

### 5.2.2.1 Specification

```

bf treads % Moves around on treads
bf wheels % Moves around on wheels
bf legs % Moves around on legs

```

```
bf turn % Able to turn an angle from initial heading
bf move_forward % Able to move forward
bf move_backward % Able to move backward
bf stay_idle % Able to stay inactive
bf limited_speed % Robot limited to low speed of locomotion
bf extended_speed % Robot can perform high speed locomotion
bf basic_control % Robot is equipped with basic control (only on and off)
bf digital_control % Robot is equipped with digital valued indication of
    locomotion speed and direction
bf small_platform % Small-size platform robot
bf medium_platform % Medium-size platform robot
bf large_platform % Large-size platform robot
bf c_s_pneumatic % Pneumatic collision sensor
bf c_s_mechanical % Mechanical collision sensor
bf c_s_combination % Collision sensor is a combination of mechanical and
    pneumatic
bf sur_finder % Small Ultrasonic Range Finder
bf lcur_finder % Low-cost Ultrasonic Ranger
bf chpu_finder % Compact High Performance Ultrasonic Ranger
bf v_s_color_vision % Sensor capable of determining the color of objects
    in the robot's environment
bf black_white_vision % Black and white environmental vision system
bf primary_color_vision % Primary colors environmental vision system
basic_means_of_locomotion= treads+wheels+legs % Basic means of locomotion
speed_of_locomotion= limited_speed+extended_speed % Speed of locomotion
locomotion_ctrl_sys= basic_control+digital_control % Locomotion Control
    System
```

```

c_sensor= c_s_pneumatic + c_s_mechanical + c_s_combination % C Sensor
rng_finder=sur_finder+lcur_finder+chpu_finder % Range Finder
platform_size_sensor=small_platform.(1+c_sensor).(1+c_sensor)
                    .(1+c_sensor) + medium_platform.(1+c_sensor)
                    .(1+c_sensor).(1+c_sensor).(1+c_sensor)
                    +large_platform.(1+c_sensor).(1+c_sensor)
                    .(1+c_sensor).(1+c_sensor).(1+c_sensor)
                    % Platform size sensor
platform_size_finder=small_platform.(1+rng_finder)+medium_platform
                    .(1+rng_finder).(1+rng_finder)+large_platform
                    .(1+rng_finder).(1+rng_finder).(1+rng_finder)
                    % Platform Size Finder
basic_platform=basic_means_of_locomotion.turn.move_forward.move_backward
                .stay_idle.(1+speed_of_locomotion).(1+locomotion_ctrl_sys)
                .(1+platform_size_sensor) % Basic Platform
enhanced_obstacle_detection=basic_platform.c_sensor.(1+platform_size_finder)
                            % Enhanced Obstacle Detection
environmental_vision=enhanced_obstacle_detection.v_s_color_vision
                    .(1+black_white_vision).(1+primary_color_vision)
                    % Environmental Vision

```

### 5.2.2.2 Results

We select the model to be the *set model*. In this specification, we have duplications of features like the case of *platform\_size\_sensor*. A *small\_platform* can have one, two, or three *c\_sensors* composed with other types of sensors. As we select the *set model*, the duplication is ignored. We *listProducts* of the

*platform\_size\_sensor* and we give the result in Appendix B.

We notice, that we have only one sensor of each type of sensors regardless of the duplications in the specification. This shows differently when we select the *bag model*, and we set the maximum occurrences to five. We choose the number five because we do not have an occurrence of a feature in the specification that is more than five. For the same family, *platform\_size\_sensor*, in the *bag model* and setting the maximum number of occurrence to five, we get in the listing given in Appendix B.

We observe that the *bag model* preserved the duplications of features. Depending on the size of the platform, the total number of sensors can be either three, four, or five regardless of their types.

In the two models, we checked whether the *small\_platform* is a subfamily of the *platform\_size\_sensor* and it is both cases. However, the *c\_sensor* is not a subfamily of *platform\_size\_sensor* as it does not stand by itself in that family regardless of the model selected.

When we *listCommonality* of the family *basic\_platform* in the *set model*, we get:

```
[
{ turn, move_forward, move_backward, stay_idle }
]
```

On the other hand, the *listCommonality* of this family in the *bag model* gives:

```
[
{
  (turn,1) , (move_forward,1) , (move_backward,1) , (stay_idle,1)
}
]
```

### 5.2.3 Other Validation Remarks

We have experimented with all the provided services that the tool supports on the above two examples and many others inducing the *Bicycle* example that we used for illustration in Section 3.2.2. The examples varied in the size of the number of features, and the maximum occurrences of features. Some examples were written to be suitable for the *set model* and others for the *bag model*. We have written specifications of examples having families identical in the composition but different in the order we write their features in a specification line. We sometimes write two families in two expressions, one with duplications of features and without and verify if they are equal in a *set model* and we found that they are.

Through these walk-through examples and the others that we experimented with, we find that the tool was a useful aid to the process of feature modelling.

# Chapter 6

## Conclusion and Future Work

We have accomplished this work to fulfil the need for a tool that supports *Feature Modelling* formally and precisely, and provides a means for specification, calculus, analysis and inference on *product families*. We have given the design of *Jory* which takes into consideration bringing together the benefits of the graphical and the non-graphical FM techniques. We have also implemented the tool to be a platform independent, scalable and extendible.

In order to bring up our design and implement the tool, we needed to get a good understanding of the concepts of *product families* and *feature modelling*. We also needed to have a close look at the FM techniques, the graphical and the non-graphical and understand their benefits, capabilities and limitations, and compare them and see their similarities and differences. We also needed to know what tools are there to support feature modelling, how they are used and what sort of support they provide.

We have chosen PFA as the kernel for implementing our tool since PFA is based on mathematics and provides a formal way to handle feature models. In addition to PFA's ability to handle feature models mathematically, it can work as a communication bridge between the graphical feature models that facilitate

the translation from one notation to another and from a graphical feature model to a specification in PFA. This makes *Jory*, once completed, a tool that serves and connects the groups that use *feature modelling*.

We have taken into consideration the need to handle large and critical systems. For that reason, we looked into a solution and we decided to build the tool based on BDDs. The BDDs are able to handle large number of features efficiently. We have also chosen Haskell for evaluating and processing the user specifications. We have also advised to use the syntax-based translation to enable users who prefer the graphical FM techniques to translate from one technique to another or from any technique to PFA and vice versa.

In all the parts of the thesis, we have taken into consideration the importance of giving an insight to the reader by giving tangible examples; examples that can be used on the tool. The examples are mainly put for illustrating the concepts, the way we perform *feature modelling*, and for highlighting the capabilities of the tool. In the remaining, we show our contribution and point to future work.

## 6.1 Contribution

The main contribution of our work is to provide the design and the implementation of four layers (among five layers) of a tool for *feature modelling* based on mathematics. The design and the implementation aim to bring together the benefits of all the feature models and extend them with formal and mathematical capabilities for specification, calculus, analysis, and inference.

The design is tailored in a layered architecture for the layers to be managed, enhanced or replaced independently. This puts into consideration the importance of making the tool platform-independent, adopt the best technologies and extend the tool with future concepts in feature modelling.



We have implemented PFA based on BDDs to provide efficient handling of large and critical systems. We have implemented two useful models which are the *set* and the *bag* models using BDDs. We use BDDs to encode the sets or the bags of features and we aggregate them as sets of sets or sets of bags. Using BDDs, we have built the basic operations in PFA, built the essential functions in sets and bags from which we constructed some more utility functions.

The design also incorporates a layer for translation. We proposed the syntax-based translation where the user provides the specification either in PFA or in *Graphviz Dot* language to be translated from one desired format to another, be it in PFA or a graphical notation.

The proposed design of the tool envisions future useful capabilities and extensions as we see it at this point of time. We have implemented most of the layers and the main parts of the tool. We have implemented the BDD layer, the set and the bag models in the concrete models layer, the term evaluation layer, and the user interface layer. The implementation of further concrete models and the proposed translation layer are subjects of future work.

The tool as it is designed and implemented, provides a powerful and precise means for handling feature models formally.

## 6.2 Future Work

The proposed tool is built to scale and extend. It is built to incorporate many needed utilities to facilitate feature modelling in practice. The tool can be extended in many ways. In the following, we give some plausible future work extensions of the tool.

The syntax-based translation technique is very useful. It can be implemented by writing down the syntax of every graphical FM technique in *Graphviz Dot*

language. It can be obtained from the tables we provide in Appendix A. The syntax of PFA is also required. YACC or any similar library can be utilized to perform the translation.

It will be useful if the user can write the specification in some macro language terms. This would make the specification shorter. For example, instead of writing  $(1 + x)$  to express that  $x$  is optional, we can write  $opt(x)$ . Macros can be also written for other expressions like the constraints.

Constraints are taken into consideration in the implementation, however, they are not implemented yet. This is part of the the future work. The constraints include *requires*, *excludes* and *implies*.

There is also the extension of merging specifications together in one specification. This specification can be written in separate files in PFA or extracted from a graphical notation in Graphviz Dot language.

The tool can be extended with a graphical editor to load or create graphical feature models directly. The graphical editor can be used for the input, modification, and for connecting with the translation layer directly to produce a corresponding PFA specification file. The graphical editor can be useful also for viewing the feature models as whole or partially. It can be used to highlights certain parts of the feature model or the specification, zoom in, zoom out, search, calculate and analyse.

These above extensions give a broad vision of how the design and the implementation of the tool establishes a solid ground for a formal, precise and productive feature modelling tool. This is a wide-perspective extensible platform for handling product families.

# Appendix A

## FM Techniques Notations and their corresponding PFA Expressions

This is to illustrate the FM techniques notations and give the corresponding PFA terms. This should give the reader an insight with an example for each graph notation side by side with PFA expression. We illustrate FODA in A.1, FORM in A.2, Riebisch's Technique in A.5, FeatuRSEB in A.3, van Gorp's Technique in A.4, and PLUSS in A.6. We avoid the repetition of notations when possible, as it is understood from our discussion of the evolution of FM techniques and the details given in Section 2.1.

In Table A.7, we summarize the FM graph notations and give their corresponding PFA terms.

Table A.1: FODA elements and the corresponding PFA expressions

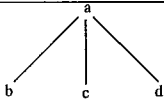
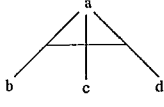
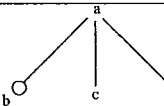
Expression	Graph	Notes	in PFA terms
$a$ consists of $b, c$ and $d$		AND-composition	$a = b.c.d$
$a$ is an XOR of $b, c$ and $d$		XOR-decomposition :similar to AND-composition with a line crossing the edges.	$a = b + c + d$
textual constraint: <i>in a product <math>a</math>, <math>b</math> requires <math>c</math> :</i>	in text	in a product $a$ , $b$ always requires the feature $c$ .	$b \xrightarrow{a} c$
textual constraint: <i><math>a</math> mutex <math>b</math> :</i>	in text	in a product $a$ , $b$ cannot coexist with $c$	$b.c \xrightarrow{a} 0$
optional		a blank circle. The feature $b$ is optional.	$a = (b + 1).c.d$

Table A.2: FORM elements and the corresponding PFA terms

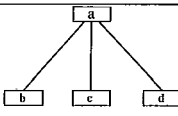
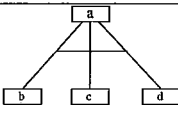
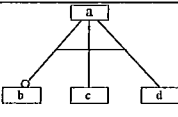
Expression	Graph	Notes	in PFA terms
$a$ is composed of $b, c$ and $d$		AND-composition	$a = b.c.d$
$a$ is an XOR of $b, c$ and $d$		XOR-decomposition	$a = b + c + d$
$b$ is an optional feature		a blank circle. The feature $a$ is optional.	$a = (b + 1).c.d$

Table A.3: FeaturSEB elements and the corresponding PFA terms

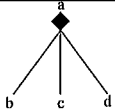
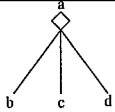
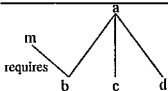
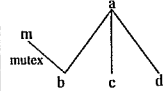
Expression	Graph:	Notes	in PFA Terms
$a$ is either $b$ , $c$ or $d$		<i>OR-decomposition</i>	$a = b + c + d + (b \cdot c) + (b \cdot d) + (c \cdot d) + (b \cdot c \cdot d)$
$a$ is an XOR of $b$ , $c$ and $d$		<i>XOR-composition</i>	$a = b + c + d$
graphical constraint: in a product $a$ , $b$ requires $m$ :		in $a$ , $b$ always requires $m$	$b \xrightarrow{a} m$
graphical constraint: in a product $a$ , mutex $b$ :		in $a$ , $b$ cannot co-exists with $m$	$b.m \xrightarrow{a} 0$

Table A.4: van Gorp elements and the corresponding PFA terms

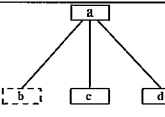
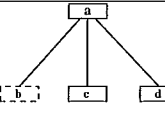
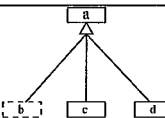
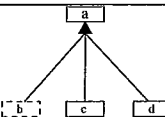
Expression	Graph	Notes	in PFA Terms
<i>Binding Times</i>		all solid-line boxes	$1 \xrightarrow{a} c.d$ or $a = c.d$ . The reconciliation is: $1 \xrightarrow{a} c.d \wedge 1 \xrightarrow{a'} b \Rightarrow 1 \xrightarrow{a.a'} b.c.d$ where, $a'$ is another view of the product.
<i>External Features</i>		dashed line boxes	$1 \xrightarrow{a'} b$ where $a'$ is the external view of $a'$
$a.a'$ is an <i>XOR</i> of $b, c$ and $d$		blank triangle	$a.a' = b + c + d \wedge b.c \xrightarrow{a.a'} 0 \wedge b.d \xrightarrow{a.a'} 0 \wedge c.d \xrightarrow{a} 0$
$a.a'$ is an <i>OR</i> of $b$ and $c$		black triangle	$a.a' = b + c + d \wedge \neg(a.a' \leq 1)$



Table A.5: Riebisch elements and the corresponding PFA terms

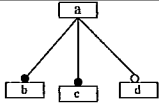
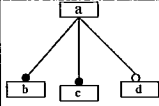












































Expression	Graph	Notes	in PFA Terms
Mandatory Feature		the edge end describes which node is <i>mandatory</i> ; <i>a</i> and <i>b</i> are.	$a = b.c.(d + 1)$
Optional Feature		the edge end describes which node is <i>optional</i> ; <i>d</i> is.	$a = b.c.(d + 1)$

Table A.6: PLUSS elements and the corresponding PFA terms

Expression	Graph	Notes	in PFA Terms
<i>Mandatory Feature</i>		black circle: $c$ is.	$a = (b + 1).c$
<i>Optional Feature</i>		blank circle: $b$ is.	$a = (b + 1).c$
<i>Single Adapter: XOR-decomposition</i>		$a$ circled S	$a = b + c + d$
<i>Multiple Adapter: OR-decomposition</i>		$a$ circled M	$a = b + c + d +$ $(b \cdot c) + (b \cdot d) +$ $(c \cdot d) + (b \cdot c \cdot d)$

Table A.7: Feature Modelling Notations and the Corresponding PFA Terms

Expression	FODA	FORM	FOPLE	FeatuRSEB	GP	van Gorp	Riebisch	PLUSS	in PFA Term
Feature	$a$			$a$	$a$	$a$	$a$	$a$	$a$
Mandatory							 , & multiplicities		in a product $p$ , $p = a$
Optional							 , & multiplicities		in a product $p$ , $p = a + 1$
AND Composed- of							UML Multiplicities		$a = b.c.d$
XOR - Alternative							UML Multiplicities		$a = b+c+d$
OR Choice	not supported	not supported	not supported				UML Multiplicities		$a = b + c + d + (b \cdot c) + (b \cdot d) + (c \cdot d) + (b \cdot c \cdot d)$
Requires	textual	textual	textual		textual				in a product $p$ , $a \xrightarrow{p} b$
Excludes	textual	textual	textual		textual				in a product $p$ , $a.b \xrightarrow{p} 0$



# Appendix B

## Robot Example Results

The listing of *listProducts* of the *platform\_size\_sensor* in the *set model*.

```
[
{ large_platform }
{ large_platform, c_s_combination }
{ large_platform, c_s_mechanical }
{ large_platform, c_s_mechanical, c_s_combination }
{ large_platform, c_s_pneumatic }
{ large_platform, c_s_pneumatic, c_s_combination }
{ large_platform, c_s_pneumatic, c_s_mechanical }
{ large_platform, c_s_pneumatic, c_s_mechanical, c_s_combination }
{ medium_platform }
{ medium_platform, c_s_combination }
{ medium_platform, c_s_mechanical }
{ medium_platform, c_s_mechanical, c_s_combination }
{ medium_platform, c_s_pneumatic }
{ medium_platform, c_s_pneumatic, c_s_combination }
{ medium_platform, c_s_pneumatic, c_s_mechanical }
{ medium_platform, c_s_pneumatic, c_s_mechanical, c_s_combination }
```

```

{ small_platform }
{ small_platform, c_s_combination }
{ small_platform, c_s_mechanical }
{ small_platform, c_s_mechanical, c_s_combination }
{ small_platform, c_s_pneumatic }
{ small_platform, c_s_pneumatic, c_s_combination }
{ small_platform, c_s_pneumatic, c_s_mechanical }
{ small_platform, c_s_pneumatic, c_s_mechanical, c_s_combination }
]

```

Listings of *listProducts* of the *platform\_size\_sensor* in the *bag model*:

```

[
  {
    (large_platform,1)
  }
  {
    (large_platform,1) , (c_s_combination,1)
  }
  {
    (large_platform,1) , (c_s_combination,2)
  }
  {
    (large_platform,1) , (c_s_combination,3)
  }
  {
    (large_platform,1) , (c_s_combination,4)
  }
  {

```

```
(large_platform,1) , (c_s_combination,5)
}
{
(large_platform,1) , (c_s_mechanical,1)
}
{
(large_platform,1) , (c_s_mechanical,1) , (c_s_combination,1)
}
{
(large_platform,1) , (c_s_mechanical,1) , (c_s_combination,2)
}
{
(large_platform,1) , (c_s_mechanical,1) , (c_s_combination,3)
}
{
(large_platform,1) , (c_s_mechanical,1) , (c_s_combination,4)
}
{
(large_platform,1) , (c_s_mechanical,2)
}
{
(large_platform,1) , (c_s_mechanical,2) , (c_s_combination,1)
}
{
(large_platform,1) , (c_s_mechanical,2) , (c_s_combination,2)
}
{
(large_platform,1) , (c_s_mechanical,2) , (c_s_combination,3)
```

```

}
{
  (large_platform,1) , (c_s_mechanical,3)
}
{
  (large_platform,1) , (c_s_mechanical,3) , (c_s_combination,1)
}
{
  (large_platform,1) , (c_s_mechanical,3) , (c_s_combination,2)
}
{
  (large_platform,1) , (c_s_mechanical,4)
}
{
  (large_platform,1) , (c_s_mechanical,4) , (c_s_combination,1)
}
{
  (large_platform,1) , (c_s_mechanical,5)
}
{
  (large_platform,1) , (c_s_pneumatic,1)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_combination,1)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_combination,2)
}

```



```
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_combination,3)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_combination,4)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1) , (c_s_combination,1)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1) , (c_s_combination,2)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1) , (c_s_combination,3)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,2)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,2) , (c_s_combination,1)
}
{
  (large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,2) , (c_s_combination,2)
}
{
```

```

(large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,3)
}
{
(large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,3) , (c_s_combination,
}
{
(large_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,4)
}
{
(large_platform,1) , (c_s_pneumatic,2)
}
{
(large_platform,1) , (c_s_pneumatic,2) , (c_s_combination,1)
}
{
(large_platform,1) , (c_s_pneumatic,2) , (c_s_combination,2)
}
{
(large_platform,1) , (c_s_pneumatic,2) , (c_s_combination,3)
}
{
(large_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,1)
}
{
(large_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,1) , (c_s_combination,
}
{
(large_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,1) , (c_s_combination,

```

```
}  
{  
  (large_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,2)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,2) , (c_s_combination,1)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,3)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,3)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,3) , (c_s_combination,1)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,3) , (c_s_combination,2)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,3) , (c_s_mechanical,1)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,3) , (c_s_mechanical,1) , (c_s_combination,1)  
}  
{  
  (large_platform,1) , (c_s_pneumatic,3) , (c_s_mechanical,2)  
}
```

```
{
  (large_platform,1) , (c_s_pneumatic,4)
}

{
  (large_platform,1) , (c_s_pneumatic,4) , (c_s_combination,1)
}

{
  (large_platform,1) , (c_s_pneumatic,4) , (c_s_mechanical,1)
}

{
  (large_platform,1) , (c_s_pneumatic,5)
}

{
  (medium_platform,1)
}

{
  (medium_platform,1) , (c_s_combination,1)
}

{
  (medium_platform,1) , (c_s_combination,2)
}

{
  (medium_platform,1) , (c_s_combination,3)
}

{
  (medium_platform,1) , (c_s_combination,4)
}

{
```

```
(medium_platform,1) , (c_s_mechanical,1)
}
{
(medium_platform,1) , (c_s_mechanical,1) , (c_s_combination,1)
}
{
(medium_platform,1) , (c_s_mechanical,1) , (c_s_combination,2)
}
{
(medium_platform,1) , (c_s_mechanical,1) , (c_s_combination,3)
}
{
(medium_platform,1) , (c_s_mechanical,2)
}
{
(medium_platform,1) , (c_s_mechanical,2) , (c_s_combination,1)
}
{
(medium_platform,1) , (c_s_mechanical,2) , (c_s_combination,2)
}
{
(medium_platform,1) , (c_s_mechanical,3)
}
{
(medium_platform,1) , (c_s_mechanical,3) , (c_s_combination,1)
}
{
(medium_platform,1) , (c_s_mechanical,4)
```

```

}
{
  (medium_platform,1) , (c_s_pneumatic,1)
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_combination,1)
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_combination,2)
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_combination,3)
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1)
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1) , (c_s_combination
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1) , (c_s_combination
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,2)
}
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,2) , (c_s_combination
}

```

```
{
  (medium_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,3)
}

{
  (medium_platform,1) , (c_s_pneumatic,2)
}

{
  (medium_platform,1) , (c_s_pneumatic,2) , (c_s_combination,1)
}

{
  (medium_platform,1) , (c_s_pneumatic,2) , (c_s_combination,2)
}

{
  (medium_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,1)
}

{
  (medium_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,1) , (c_s_combination,1)
}

{
  (medium_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,2)
}

{
  (medium_platform,1) , (c_s_pneumatic,3)
}

{
  (medium_platform,1) , (c_s_pneumatic,3) , (c_s_combination,1)
}

{
```

```
(medium_platform,1) , (c_s_pneumatic,3) , (c_s_mechanical,1)
}
{
(medium_platform,1) , (c_s_pneumatic,4)
}
{
(small_platform,1)
}
{
(small_platform,1) , (c_s_combination,1)
}
{
(small_platform,1) , (c_s_combination,2)
}
{
(small_platform,1) , (c_s_combination,3)
}
{
(small_platform,1) , (c_s_mechanical,1)
}
{
(small_platform,1) , (c_s_mechanical,1) , (c_s_combination,1)
}
{
(small_platform,1) , (c_s_mechanical,1) , (c_s_combination,2)
}
{
(small_platform,1) , (c_s_mechanical,2)
```



```
}  
  
{  
  (small_platform,1) , (c_s_mechanical,2) , (c_s_combination,1)  
}  
  
{  
  (small_platform,1) , (c_s_mechanical,3)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,1)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,1) , (c_s_combination,1)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,1) , (c_s_combination,2)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,1) , (c_s_combination,1)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,1) , (c_s_mechanical,2)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,2)  
}
```

```
{  
  (small_platform,1) , (c_s_pneumatic,2) , (c_s_combination,1)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,2) , (c_s_mechanical,1)  
}  
  
{  
  (small_platform,1) , (c_s_pneumatic,3)  
}  
]
```

# Bibliography

- [AC04] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In Michael G. Burke, editor, *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.
- [Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509 – 516, 1978.
- [Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pages 7–20. Springer Berlin/Heidelberg, 2005.
- [Big10] BigLever Software Inc. Gears. <http://www.biglever.com/>, 2010. (Last accessed on January 14, 2010).
- [Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
- [Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical report, Pittsburgh, PA, USA, 1992.
- [CAK<sup>+</sup>05] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. fmp and fmp2rsm: eclipse plugins for modeling features using model templates. In *OOPSLA '05*:

- Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 200–201, New York, NY, 2005. ACM Press.
- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
- [Cza98] Krzysztof Czarnecki. *Generative Programming, Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, October 1998.
- [DGR07] Deepak Dhungana, Paul Grnbacher, and Rick Rabiser. Decisionking: A flexible and extensible tool for integrated variability modeling. In *1st International Workshop on Variability Modelling of Software-intensive Systems*, pages 119–128, 2007.
- [DKDK02] A. Van Deursen, P. Klint, Arie Deursen, and Paul Klint. Domain-specific language design requires feature descriptions. Technical report, 2002.
- [DS06] Olfa Djebbi and Camille Salinesi. Criteria for comparing requirements variability modeling notations for product lines. In *Proceedings of the Fourth International Workshop on Comparative Evaluation in Requirements Engineering (CERE’06)*, pages 20 – 35, Washington, DC, September 2006. IEEE Computer Society.
- [DSF07] Olfa Djebbi, Camille Salinesi, and Gauthier Fanmuy. Industry survey of product lines management tools: Requirements, qualities and

- open issues. *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 301–306, 2007.
- [EBB] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In *Software Product Lines - 9th International Conference, SPLC 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 33–44.
- [FAC07] Alain Forget, Dave Arnold, and Sonia Chiasson. Case-fx: feature modeling support in an OO case tool. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 803–804, New York, NY, USA, 2007. ACM.
- [Gee03] Geert Janssen. A consumer report on bdd packages. In *SBCCI '03: Proceedings of the 16th symposium on Integrated circuits and systems design*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.
- [Gen10] Generative Programming. AmiEddi Tool. <http://www.generative-programming.org/>, 2010. (Last accessed on November 26, 2009).
- [GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, pages 76–85, Washington, DC, USA, 1998. IEEE Computer Society.
- [HKM06] Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature Algebra. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*

- 2006: *Formal Methods*, volume 4085 of *Lecture Notes in Computer Science series*, pages 300–315, 14th International Symposium on Formal Methods, McMaster University, Hamilton, Ontario, Canada, August 21 - 27 2006. Springer.
- [HKM08] Peter Höfner, Ridha Khedri, and Bernhard Möller. Algebraic view reconciliation. In *6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 85–94. Cape Town, South Africa, November 10-14, 2008.
- [HKM09] Peter Höfner, Ridha Khedri, and Bernhard Möller. An Algebra of Product Families. *Software and Systems Modeling*, 2009.
- [HM85] M. C. B. Hennesy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.
- [Hon10] Honeywell. DoME. <http://www.htc.honeywell.com/dome>, 2010. (Last accessed January 14, 2010).
- [IBM10] IBM. DOORS T-REK. <http://www-01.ibm.com/software/awdtools/doors/productline/>, 2010. Last accessed February 23, 2010.
- [KCH<sup>+</sup>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, November 1990.
- [KDn06] Timo Käkölä and Juan C. Dueñas. *Software Product Lines - Research Issues in Engineering and Management*. Springer, 2006.

- [KKL<sup>+</sup>] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168.
- [KLD02] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, July/August 2002.
- [Kot05] Koteswar Rao Kollu. Evaluating The PLUSS Domain Modeling Approach by Modeling the Arcade Game Maker Product Line. Master’s thesis, Umeå University, SE-901 87 UMEÅ, SWEDEN, 21 June 2005.
- [Kru93] R. Krut. Integrating 001 tool support into the feature-oriented domain analysis methodology. Technical report, Software Engineering Institute, Carnegie Mellon University, May 1993.
- [KSA<sup>+</sup>06] Kyo C. Kang, Minseok Seo, Miyoung Ahn, Yeop Chang, Hyejung Kim, and Kyungseok Kim. Asadal: a tool system for co-development of software and test environment based on product line engineering. *Proceedings of the 28th international conference on Software engineering*, pages 783–786, 2006.
- [LC10] AT and T Research Labs and Contributors. Graphviz: A graph visualization software. <http://www.graphviz.org/>, 2010. Last accessed on Feb 2, 2010.
- [Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

- [LN10] Jørn Lind-Nielsen. Buddy BDD Library. <http://sourceforge.net/projects/buddy/>, 2010. (Last accessed on Feb 2, 2010).
- [Pet71] S. R. Petrick. On the use of syntax-based translators for symbolic and algebraic manipulation. In *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 224–237, New York, NY, USA, 1971. ACM Press.
- [Pur10] Pure Systems. Pure::Variants. <http://www.pure-systems.com/3.0.html>, 2010. (Last accessed on November 26, 2009).
- [RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending Feature Diagrams with UML Multiplicities. *6th Conference on Integrated Design & Process Technology (IDPT 2002)*, Pasadena, California, USA, 2002.
- [Req10] RequiLine. RequiLine Tool. <http://www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline/>, 2010. (Last accessed on November 26, 2009).
- [Rob03] Silva Robak. Feature Modeling Notations for System Families. In *International Workshop on Software Variability Management (SVM)*, pages 58–62, Portland, Oregon, 2003.
- [RZS09] Andrei Rimsa, Luis E. Zárate, and Mark A. Song. Evaluation of different bdd libraries to extract concepts in fca — perspectives and limitations. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 367–376, Berlin, Heidelberg, 2009. Springer-Verlag.



- [SGI10] SGI. C++ STL library. <http://www.sgi.com/tech/stl/>, 2010. (Last accessed on Feb 2, 2010).
- [SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.
- [Sou10] Source Forge. CaptainFeature Tool. <http://sourceforge.net/projects/captainfeature/>, 2010. (Last accessed on November 26, 2009).
- [SSP<sup>+</sup>00] Douglas Stuart, Wonhee Sull, Steve Pruitt, Deborah Cobb, Fred Waskiewicz, and T. W. Cook. The SSEP toolset for product line development: an xml-based, architecture-centric approach. In *Proceedings of the first conference on Software product lines : experience and research directions*, pages 413–435, Norwell, MA, USA, 2000. Kluwer Academic Publishers.
- [Str04] Detlef Streitferdt. *Family-oriented requirements engineering*. PhD thesis, Technical University Ilmenau, 2004.
- [Uni10] University of Duisburg-Essen. VarMod. <http://www.sse.uni-due.de/wms/en/index.php?go=139>, 2010. (Last accessed on January 14, 2010).
- [Utr10] Utrecht University. Parsec Library, 2010. (Last accessed on Feb 2, 2010).
- [vdML04] Thomas von der Maßen and Horst Lichter. Requiline: A requirements engineering tool for software product lines. In *Software Product-*

- Family Engineering*, Volume 3014/2004, pages 168–180. Springer Berlin / Heidelberg, 2004.
- [vGBS01] J. van Gorp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 45, Washington, DC, 2001. IEEE Computer Society.
- [XFe10] XFeature. XFeature. <http://www.pnp-software.com/XFeature/>, 2010. (Last accessed on November 26, 2009).