

East West University

Analysis of Feature Tree using Miranda

Ferdous Hussain Badhan

Id: 2011-3-60-014

&

Md Shariful Islam

Id: 2014-2-60-048

A thesis submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in
Computer Science and Engineering



Department of Computer Science and Engineering

December 2017

Declaration

I, hereby, declare that the work presented in this thesis is the outcome of the investigation performed by me under the supervision of **Dr. Shamim H. Ripon**, Associate Professor, Department of Computer Science and engineering, East West University. I also declare that no part of this thesis/project has been or is being submitted elsewhere for the award of any degree or diploma.

.....
(Dr. Shamim H. Ripon)

Supervisor

Associate Professor,
Department of CSE
East West University

.....
(Ferdous Hussain Badhan)

ID: 2011-3-60-014

.....
(Dr. Ahmed Wasif Reza)

The chairperson

Associate Professor,
Department of CSE
East West University

.....
(Md Shariful Islam)

ID: 2014-2-60-048

Abstract

Feature models are used to specify the variability of software product lines. We study a special subset of trees, called generalised feature trees, and show how they can be used to compute properties of the corresponding software product lines. We introduce our paper the concept of generalised feature trees, which are feature trees where features can have multiple occurrences. It is shown how an important class of feature models can be transformed into generalised feature trees. We present algorithms which, after transforming a feature model to a generalised feature tree, compute properties of the corresponding software product line. We discuss the computational complexity of these algorithms and provide executable specifications in the functional programming language Miranda.

Acknowledgements

First of all thanks to our Supervisor **Dr. Shamim H Ripon** for providing us this opportunity to test our skill in the best possible manner. He enlightened, encouraged and provided us with ingenuity to transform our vision into reality.

This project would not be possible without the help of our project supervisor, **Dr. Shamim H Ripon**, Associate Professor, Department of Computer Science and Engineering, East West University. He helps us to understand all the matters easily which make us to create this project. We would like to express our sincere and deep regards to him.

Lastly, we both are really thankful to Almighty **ALLAH**. So at the end deliberately we want to pay tribute to our parents. We call them “Our Heroes, Our Mentors”. These are the people that **ALLAH** has used to discover nature and deepen our academic career. There are numerous other people too who have shown me their constant support and friendship in various ways, directly or indirectly related to my/our academic life. We will remember them in our heart and hope to find a more appropriate place to acknowledge them in the future.

Md Ferdous Hussain Badhan

2011-3-60-014

Md Shariful Islam

2014-2-60-048

December, 2017

Table of Contents

Declaration	i
Abstract	ii
Acknowledgement	iii
Chapter 1: Introduction	
1.1 Introduction and Motivation.....	1
1.2 Objectives.....	2
1.3 Contribution.....	3
1.4 Outline.....	4
Chapter 2 : Background	
2.1 Software Product Line.....	5
2.2 Feature Model.....	7
2.3 Feature Tree Representation.....	9
2.4 Feature Modeling Notation.....	9
2.5 Example of Feature Tree.....	14
2.6 Miranda.....	15
2.7 Installing Miranda under Cygwin.....	16
2.8 Type System	22
2.9 How does type checking.....	23
Chapter 3: Analysis Rules of Feature Model	
3.1 Feature Model.....	27
3.2 Analysis Operations on Feature Model.....	31
Chapter 4: Analysis of Feature Model Using Miranda	
4.1 Software Product Line.....	37
4.2 Model.....	38
4.3 Diagram.....	38
4.4 Configuration.....	38
4.5 Feature Model Notation.....	38
4.6 Basic Feature Model.....	39
4.7 Scenario of Feature Model.....	40
4.8 Feature Models.....	47
4.9 Generalized Feature Tree.....	48
4.10 Analysis of Feature Model.....	49
4.11 Minimal Set of Conflicting Constraints.....	52
4.12 Explanation of Dead Features.....	53
4.13 Example of Feature Tree Using Miranda	55
4.14 Constraints.....	57
4.15 Computational Complexity	58
Chapter 5: Summary and Feature Work	
5.1 Summary	60
5.2 Future Work.....	60
Bibliography.....	61

Chapter 1

Introduction

1.1 Introduction and Motivation

A Feature Tree (sometimes also known as a Feature Model or Feature Diagram) is a hierarchical diagram that visually depicts the features of a solution in groups of increasing levels of detail. Feature Trees are great ways to summarize the features that will be included in a solution and how they are related in a simple visual manner. In the Feature Tree, some features may be flagged as mandatory, some optional, and some as mutually exclusive. In general, the “features” in a feature tree could include functional features (hardware, software), non-functional features (performance or other criteria), or even parameters (the same feature at varying levels of capability or cost for example) [1]. In most cases however, the features of Feature Tree are mostly kept at a summary level. They are most commonly used for planning the overall feature set of a single solution (scope) or product that will be evolved over time (say through multiple iterations of development, or multiple releases to the marketplace) or for defining the differing features that will be included in a product line (for example, to define the differences between Windows 7 Home, Pro, and Ultimate or for different trim levels in a range of automobiles) [2].

The report introduces the overall description of “Analysis of Feature tree using Miranada”. In software development, a **feature model** is a compact representation of all the products of the Software Product Line (SPL) in terms of "features". Software Product Line (SPL) is a set

of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. It will explain the Feature Tree by the rules of Logical representation. Software product lines centralize upon the idea of designing and implementing a family of systems to produce qualitative applications in a domain, promote large scale reuse and reduce development costs [3].

Feature models are used to specify the variability of software product lines [1,2]. we present an example and discuss the computational complexity of our approach. Throughout the paper, we present executable specifications of all algorithms in the functional programming language Miranda.

1.2 Objectives

Based on the logical rules, the analysis of a feature tree using is introduced. we study a special subset of those trees, called generalised feature trees, and show how they can be used to compute properties of the corresponding software product lines. We define a generalised feature tree (GFT) to be a feature tree whose features, instead of being required to be all distinct, satisfy the following two restrictions:

Restriction 1: when two nodes of a GFT have the same feature, they belong to different sub trees of an Xor node.

Restriction 2: for each node of a GFT, all sub trees have disjoint semantics.

We discuss here some of the topic of feature tree analysis. These are given below:

- Software Product Line (SPL)
- Feature Tree
- Logical Notation of Feature Tree
- Miranda Language and It's Function
- Representation of Feature tree with Miranda
- Verification of Dead Features, False Optional
- Determination of List of Products

In the next section we briefly describe the feature models we consider in this paper. we introduce the concept of generalised feature tree and describe algorithms which deal with commitment to a feature and deletion of a feature of a GFT. In section 4 we describe how a large class of feature models can be mapped to equivalent GFTs. In next we show how this mapping can be used for the analysis of feature models. Then we present an example and in next section we discuss the computational complexity of our approach. Throughout the paper, we present executable specifications of all algorithms in the functional programming language Miranda [1].

1.3 Contribution

The aim of this chapter is to provide We present algorithms which, after transforming a feature model to a generalised feature tree, compute properties of the corresponding software product line. We discuss the computational complexity of these algorithms and provide executable specifications in the functional programming language Miranda.

Most of the properties of SPL feature tree modeling have been worked properly. Though there are some drawbacks in the system. So we analysis here the feature tree that are give bellow

- The first step of the analysis is the computation of an equivalent GFT
- Existence of products
- Dead features
- Number of products
- List of all products
- Products which contain a given set of features
- Minimal set of conflicting constraints

1.4 Outline

Chapter 2: In this chapter we will glimpse the background that is about Software Product Line, Feature Model, Logical Notation of Feature and introduce

Chapter 3: Here the Logical Representation of Feature Model will be chronicled by the Logical representation of Features through the six logical rules and Feature analysis.

Chapter 4: Here we implement a Feature Tree in with logical notation. Then we add some constraints and analysis that feature tree using Miranda.

Chapter 5: The rest of the things of the report will be represented in this chapter. That is all about the conclusion part of this paper and we will also discuss the achievement from this project and the future work with this project to develop it more.

Chapter 6: It is all about the user Manual of the whole system.

Chapter 7: Last but not list that is the guidance of reference and the codes full of appendix.

Chapter 2

Background

2.1 Software product lines(SPL)

Software product lines, or software product line development, refer to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production. [4]

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

Software product lines are emerging as a viable and important development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers. Software product line engineering can also enable rapid market entry and flexible response, and provide a capability for mass customization.

We are working to make software product line . practice a dependable low-risk high-payoff practice that combines the necessary business and technical approaches to achieve success. If you would like to gain expertise in these practices, see **training** in product lines.

2.1.1 Benefits

Product lines can help organizations overcome the problems caused by resource shortages. Organizations of all types and sizes have discovered that a product line strategy, when skillfully implemented, can produce many benefits and ultimately give the organizations a competitive edge.

Example organizational benefits include:

- Improved productivity by as much as 10x
- Increased quality by as much as 10x
- Decreased cost by as much as 60%
- Decreased labor needs by as much as 87%
- Decreased time to market (to field, to launch) by as much as 98%
- Ability to move into new markets in months, not years

While early software product line methods at the genesis of the field provided the best software engineering improvement metrics seen in four decades, the last generation of software product line methods and tools are exhibiting even greater improvements. New generation methods are extending benefits beyond product creation into maintenance and evolution, lowering the overall complexity of product line development, increasing the scalability of product line practice with orders of magnitude less time, cost and effort. Domain and application engineering are the two main phases of SPL development [4].

2.1.2 For example

- Cummins, Inc., was able to field more than 1000 separate products based on just 20 software builds. They can build and integrate the software for a new diesel engine in about a week, whereas before, it took a year. Their production capability allowed them to quickly enter and dominate the industrial diesel engine market.

- The U.S. National Reconnaissance Office commissioned a software product line of satellite ground control systems from Raytheon and enjoyed a 10x quality improvement and a 7x productivity improvement as a result.
- Celsius Tech Systems was able to decrease their software staff from 210 to around 30, while turning out more, larger, and more complex ship command and control system products. The product line approach let them change the hardware-to-software ratio for their systems from 35:65 to 80:20.
- Nokia was able to increase their production of mobile phones from 5 to 10 new models per year to over 30 new models per year.
- Hewlett Packard reported a 400% productivity improvement and a 2-7x time-to-market improvement for a product line of printers.
- Motorola saw a 4x cycle-time improvement in its product line of pagers.

2.2 Feature Model

A SPL is a family of related programs. When the units of program construction are features increments in program functionality or development every program in an SPL is identified by a unique and legal combination of features, and vice versa.

Feature models were first introduced in the Feature-Oriented Domain Analysis (FODA) method by Kang in 1990. Since then, feature modeling has been widely adopted by the software product line community and a number of extensions have been proposed [5].

Feature modeling is the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model. A feature model represents the common and the variable features of concept instances and the dependencies between the variable features. Model represents the intention

of a concept, whereas the set of instances it describes is referred to as the extension of the concept. A feature model consists of a feature diagrams and some additional information such as short semantic description of each feature, rationale for each feature, constraints, default dependency rules etc. A feature diagram consists of a set of nodes, a set of directed edges, and a set of edge decorations. The nodes and the edges form a tree. he edge decorations are drawn as arcs connecting subsets or all of edges originating from the same node.[6]

A mandatory feature is included in the description of a concept instance if and only if its parent is included in the description of the concept. If the parent of a mandatory feature is optional, the mandatory feature should not be part of the description. A mandatory feature node is pointed to by a simple edge optionally ending with a filled circle An optional feature may be included in the description of a concept instance if the parent is included. If the parent is not included, the optional feature cannot be included. An optional feature node is pointed to by a simple edge ending with an empty circle A concept (and similarly a feature) may have one or more sets of direct alternative features. If the parent of a set of alternative features is included, then exactly one feature from this set of alternative features is included in the description, otherwise none. The nodes of a set of alternative features are pointed to by edges connected by an arc. A concept (and similarly a feature) may have one or more sets of direct or features. If the parent of a set of or-features is included in the description of a concept instance, then any non-empty subset from the set of or-features is included in the description, otherwise none. The nodes of a set of or-features are pointed to by edges connected by a filled arc Feature modeling helps us to avoid two serious problems: First, relevant features and variation points are not included in the reusable software. Second, many features and variation points are included but never used and thus cause unnecessary complexity, development cost, and maintenance cost. Finally, the feature models provide us with an abstract (since implementation independent), concise, and explicit representation of the variability present in the software [7].

2.3 Feature Tree Representation

A feature model is a compact representation of all the products of the Software Product Line (SPL) in terms of "features". Feature models are visually represented by means of feature diagrams. Feature models are widely used during the whole product line development process and are commonly used as input to produce other assets such as documents, architecture definition, or pieces of code.

2.4 Feature modeling Notation

Current feature modeling notations may be divided into three main groups, namely:

- Basic feature models
- Cardinality-based feature models
- Extended feature models

2.4.1 Basic feature models

A Feature Model (FM) is a hierarchical arranged set of features. It represents all possible products of an SPL (Software Product Line) in a single model [7]. Every Feature is an increment in product functionality. The complete feature tree of CAD domain is illustrated. It can be used in different stages of development. Though A FM is a tree like structure so it consists of relations between a parent feature and its child features, also cross-tree constrains that are typically inclusion or exclusion statements of the form “if a feature F is included, then feature X must also be included”. The relation between a parent (variation point) features and its child feature (variants) are categorized as follows:

Mandatory

A child feature is said to be mandatory when it is required to appear when the parent feature appears. For instance, it is mandatory to have a special platform for Android mobile phone. A mandatory feature is included if its parent feature is included.

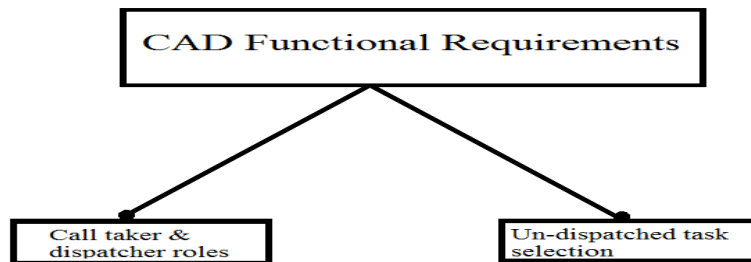


Figure 2.1: Mandatory Feature example

Optional

A child feature is said to be optional when it can or not appear when the parent features appears. For instance, it is optional to have pdf reader software in the mobile phone. An optional feature may or may not be included if its parent is included.

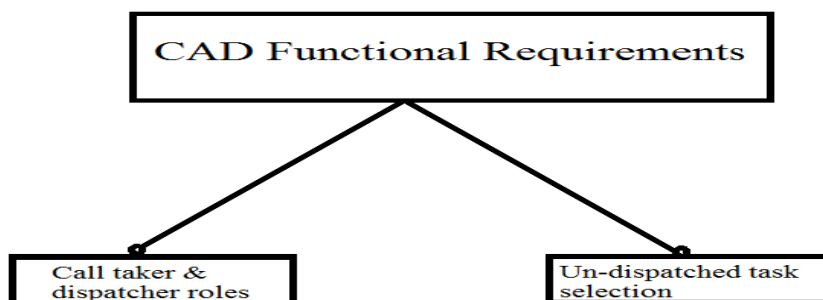


Figure 2.2: Optional Feature example

Alternative

A set of child features are said to be alternative when only one child feature can be selected when the parent feature appears. For instance a Gamer cannot select both Automatic and Manual for car control during car selection. One and only one feature from a set of alternative features are included when parent feature is included.

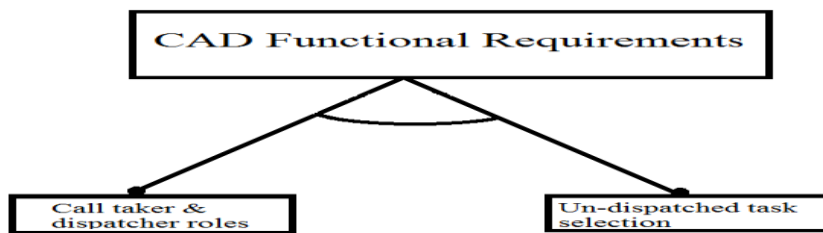


Figure 2.3: Alternative Feature example

Optional Alternative

One feature from a set of alternative features may or may not be included if parent included. One feature from a set of alternative features may or may not be included if parent included.

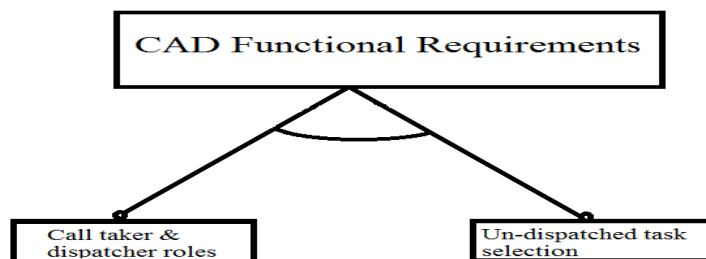


Figure 2.4: Optional Alternative Feature example

Or

A set of child features are said to have an or-relation with their parent when one or more sub features can be selected when the parent feature appears. For instance, the engine of a car can be electric, gasoline or both at the same time. At least one from a set of feature is included when parent is included.

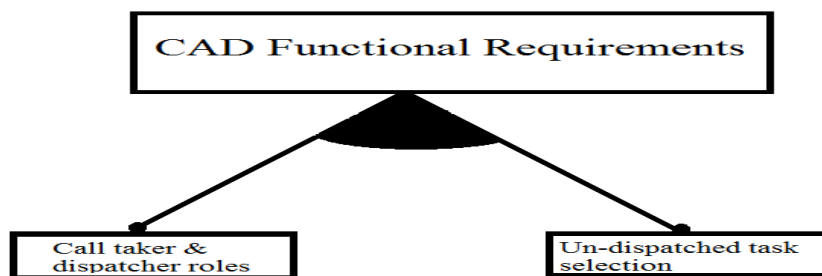


Figure 2.5: Or Feature example

Optional or

One or more optional feature may be included if the parent is included. One or more optional feature may or may not be included if the parent is included.

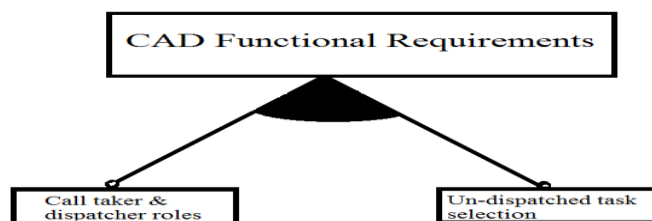


Figure 2.6: Optional Or Feature example

A feature model can be considered as a graph consists of a set of sub graphs. Each sub graph is created separately by defining a relationship between the variation point (denoted as v_i) and the variants ($v_{i,j}$) by using the expressions. The complexity of a graph construction lies in the definition of dependencies among variants. When there is a relationship between cross-tree (or cross hierarchy) variants (or variation points) we denote it as a dependency. Typically dependencies are either inclusion or exclusion: if there is a dependency between p and q, then p is included then q must be included (or excluded). Dependencies are drawn by dotted lines. [10]

In addition to the parental relationships between features, cross-tree constraints are allowed. The most common are:

- A requires B – The selection of A in a product implies the selection of B.
- A excludes B – A and B cannot be part of the same product.

2.5 Example of a Feature Tree

As an example, the figure below illustrates how feature models can be used to specify and build configurable on-line shopping systems. The software of each application is determined by the features that it provides. The root feature (i.e. E-Shop) identifies the SPL[7]. Every shopping system implements a catalogue, payment modules, security policies and optionally a search tool. E-shops must implement a high or standard security policy (choose one), and can provide different payment modules: bank transfer, credit card or both of them. Additionally, a cross-tree constraint forces shopping systems including the credit card payment module to implement a high security policy. [1]

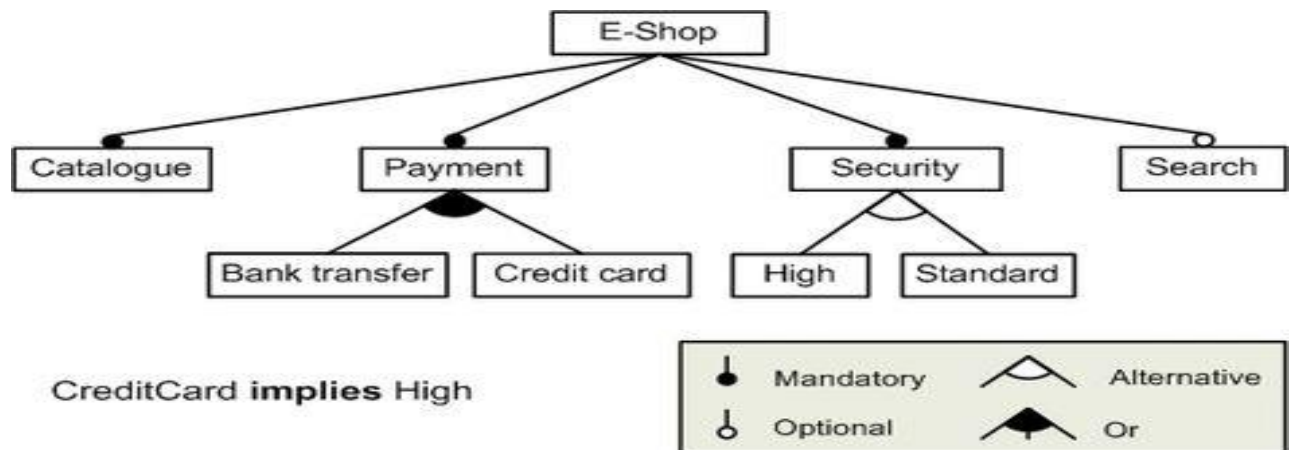


Figure 2.7: Feature Tree Representation with the help of Logical notations

2.6 Miranda

Miranda is an example of a pure functional programming language -- there are *no* side effects.

Imperative languages have variables and assignment; they are in effect abstractions of the von Neumann machine. There is no referential transparency.

In mathematics, we can substitute equals for equals:

if $x=y+z$ then you can substitute $y+z$ for x

Informally, in a referentially transparent language you can substitute equals for equals, just as in mathematics.

Interesting features of Miranda:

- truly functional
- lazy evaluation -- can deal with infinite structures
- type system -- statically typed, no type declarations needed; polymorphic
- combinator-based implementation

Haskell is a standard pure functional language, but it is much more complex than Miranda.

ML has a functional subset.

future of functional languages ...?

barriers:

- efficiency (a declining problem; also functional languages good candidates for parallel execution)
- programmer acceptance
- unnatural for some things, such as interactive I/O

2.7 Installing Miranda under Cygwin

Open a Cygwin bash window by clicking on the Cygwin icon, or from the Start menu. List the contents of your home directory by saying

```
ls
```

you should see the download file `mira-2039-i686-cygwin.tgz`. To unpack Miranda type

```
tar xzpf mira-2041-i686-Cygwin.tgz
```

This will create an installation directory `mira-2041-i686-Cygwin`. To install type

```
cd mira-2041-i686-Cygwin
sh install
cd ..
```

Note that bash allows you to complete words by pressing `tab`. To run Miranda type

```
Mira
```

There is online help information (say `/h`) and an online manual (say `/man`). To access the manual directly from a UNIX shell say

```
mira -man
```

2.7.1 Some more details

The UNIX manual page for Miranda, which gives information for installers and administrators, is accessed in the usual way

```
man mira
```

Also included in the release are `just` and `mtotex`, text formatting programs, sometimes useful in connection with Miranda literate scripts (see Miranda online manual).

These have manual pages, say `man just`, `man mtotex`.^[14]

You can safely remove the installation directory and the downloaded `.tgz` file after installation. To remove a file say `rm filename`, to remove a whole directory say `rm -rf dirname`.

You can inspect the `install` script in the installation directory to see what it is installing and where. Note in particular `miralib`, a directory containing various files which `mira` needs when it is running, which is usually placed at `/usr/lib/miralib`. For more information say `man mira`.

There is a collection of example programs, in a directory called `ex`, under the `miralib` directory. It has a `README` file in it. Within a Miranda session, saying

```
/cd <ex>
```

will take you into the examples directory. Say

```
!ls
```

to see what is in there.

2.7.2 Running Miranda

The system is available on the instructional linux cluster (`ceylon`, `fiji`, `sumatra`, `tahiti`). It is also available on `orcas` - but I'd use the linux boxes since they are faster. To run it, type `mira`

To load in an existing file named `myfile.m`, type

```
mira myfile.m
```

For a list of useful commands within Miranda type `/help`. These include the ability to edit the current program (or script, to use Miranda terminology). To do this, type `/edit`. The newly edited script is then loaded in. Other commands:

```
/man - online manual
```

```
/quit (or control d)
```

The default editor is `vi`, but you can change this by adding the following line to your `.cshrc` file:

```
setenv EDITOR /usr/local/bin/emacs
```

Miranda is an interactive tty-oriented environment (like Scheme). Examples:

```
Miranda 3+4
7
```

```
Miranda 8 : [1..10]
[8,1,2,3,4,5,6,7,8,9,10]
```

```
Miranda [1,2,3] ++ [8,9]
[1,2,3,8,9]
```

```
Miranda member [1,2,3] 8
False
```

```
To define a function:
double x = x+x
```

Function and other definitions must go in a file (script); Miranda distinguishes between definition mode (when reading from a file) and evaluation mode.

2.7.3 Data types

- numbers (both int and float): 3 4.5 3.9e10
- booleans: True False
- characters: 'a'
- lists: [1,2,3,8]

type system requires that all elements in a list must be of the same type:

```
[2, 4, 7]
[[1, 2], [4, 9]]
```

strings -- shorthand for list of chars "hi there"

```
tuples: (False, [1, 2, 3], "str")
```

functions are first class citizens

function application denoted by juxtaposition

```
neg 3
-3
member [1, 4, 6] 4
infix --
3+4
(+ ) 3 4
```

```
arithmetic
+ - * /
sin cos etc
sum, product
```

```
product [1, 2, 8]
```

2.7.4 Recursion Examples

Miranda is quite terse. See `~borning/miranda/*` on orcas for example programs (or scripts).

All the examples in these lecture notes are on `lecture.m`

```

|| recursive factorial
|| whitespace is significant - notice the use of layout in the example
rec_factorial n = 1, if n=0
                = n * rec_factorial (n-1), n>0

|| alternate version using pattern matching:
pattern_factorial 0      = 1
pattern_factorial (k+1) = (k+1) * pattern_factorial k
|| factorial using the built-in prod function
factorial n = product [1..n]
|| mapping function, like map in Scheme
my_map f [] = []
my_map f (a:x) = f a : my_map f x

my_map factorial [1,5]
my_map factorial [1..10]
my_map factorial [1..]

```

2.7.5 Currying

```

plus x y = x+y

f = plus 1

ff = (+) 1

twice f x = f (f x)

g = twice f

g 3

```

2.7.6 Block structure - lexical scoping

```

hyp x y = sqrt sum
          where sum = x*x + y*y

hyp2 x y = sqrt sum
           where sum = x2 + y2
                 where x2 = x*x
                       y2= y*y

```


2.7.7 Lazy Evaluation and Infinite Lists

```
my_if True x y = x
```

```
my_if False x y = y
```

```
my_if (x>y) 3 4
```

```
my_if (x>y) 3 (1/0)
```

```
ones = 1 : ones
```

Compare with circular list in Scheme:

```
(define ones '(1))
```

```
(set-cdr! ones ones)
```

```
ints n = n : ints (n+1)
```

```
[1..]
```

```
lets = "abc" ++ lets
```

Two prime number programs:

```
factors n = [k | k <- [1..n]; n mod k = 0]
```

```
dullprimes = filter isprime [2..]
```

```
  where
```

```
    isprime p = (factors p = [1,p])
```

```
interestingprimes = sieve [2..]
```

```
  where
```

```
    sieve (p:x) = p : sieve [n | n <- x; n mod p > 0]
```

Hamming numbers:

```
my_merge (x:a) (y:b) = x : my_merge a (y:b) , if x<y
```

```
                = y : my_merge a b , if x=y
```

```
                = y : my_merge (x:a) b , otherwise
```

```
ham = 1: my_merge ham2 (my_merge ham3 ham5)
```

```
ham2 = map (*2) ham
```

```
ham3 = map (*3) ham
```

```
ham5 = map (*5) ham
```

2.7.8 higher-order functions

```
twice f x = f (f x)
```

```
double = (*) 2
```

```
twice double 4
```

```

map f [] = []
map f (a:x) = f a : map f x
map ( (+) 1) [1..5]
argument and result types of function can be different:
map code "0123"
[48,49,50,51]

```

interesting uses:

```
member x a = or (map (=a) x)
```

```
length x = sum (map (const 1) x)
```

sum, product, and, or, concat, etc ...

could define recursively, e.g.

```

sum [] = 0
sum (a:x) = a+sum x
foldr op r = f
      where
      f [] = r
      f (a:x) = op a (f x)

```

```

sum = foldr (+) 0
product = foldr (*) 1
or = foldr (\/) False
and = foldr (&) True

```

This does not hurt efficiency. Miranda's implementation uses combinator graphs -- the first time you run the function it rewrites that portion of the graph. This is also the case for things like (1+3) and code '0'

2.7.9 Other examples:

```

my_filter f [] = []

my_filter f (a:x) = a : my_filter f x, f a
                  = my_filter f x, otherwise
reverse_args f x y = f y x

const k x = k

```

```
fix f = e
      where e = f e
```

```
double x = 2*x
const x y = x
```

2.8 Type system

Miranda uses Milner-style polymorphism (which was originally developed for type checking ML). A polymorphic function is a function with a type variable in its type [14]. In Milner-style polymorphism there are no restrictions on types that a type variable can take on (universal polymorphism).

```
3 ::
num
[1,2,3] ::
[num]

[[1,2,3]] ::
[[num]]

[] ::
[*]

neg ::
num -> num

+ ::
num -> num -> num
(note that -> is right associative)

member ::
[*] -> * -> bool

= ::
* -> * -> bool
```

run time error if = applied to functions

The type system is not always as descriptive as one would like.

For append and member, universal polymorphism is just right. It isn't what we want for functions -- but Miranda fudges this with a runtime type check. Miranda avoids the complexities of overloading by having a single type num rather than integer and float types. (Haskell's type system includes type classes, which address both of these issues at the cost of added complexity.)

Universal polymorphism doesn't work at all for object-oriented languages: there we want a type system that specifies that a variable x can hold any object that understands the + and * messages. (F-bounded polymorphism is a way to provide this.)

2.9 How does type checking work?

Variables are the same as the logic variables we used in CLP(R). The type checker uses unification (i.e. solving equality constraints over trees).

Examples:

```
double x = x+x
double::num->num
```

```
map f [] = []
map f (a:x) = f a: map f x
```

```
map::(*->**)->[*]->[**]
```

Here * and ** are *type variables*. Aside: the * convention for naming type variables is perhaps not ideal. In Haskell the type of map is written

```
(a -> b) -> [a] -> [b]
In ML it is
('a -> 'b) -> ['a] -> ['b]
Back to Miranda ...
map code "0123"
[48,49,50,51]
```

```
twice::
  (*->*)->*->*
map f [] = []
map f (a:x) = f a: map f x
foldr:: (*->**->**) ->**->[*]->**
foldr op r = f
      where
        f [] = r
        f (a:x) = op a(f x)
sum = foldr (+) 0
```

Chapter 3

Analysis rules of Feature Model

3.1 Feature Models

A feature model represents the information of all possible products of a software product line in terms of features and relationships among them. Feature models are a special type of information model widely used in software product line engineering. A feature model is represented as a hierarchically arranged set of features composed by:

1. Relationships between a parent feature and its child features.
2. Cross-tree (or cross-hierarchy) constraints that are typically inclusion or exclusion statements in the form: if feature F is included, then features A and B must also be included (or excluded).

Figure 3.1 depicts a simplified feature model inspired by the mobile phone industry. The model illustrates how features are used to specify and build software for mobile phones. The software loaded in the phone is determined by the features that it supports. According to the model, all phones must include support for calls, and displaying information in either a basic, color or high resolution screen. Furthermore, the software for mobile phones may optionally include support for GPS and multimedia devices such as camera, MP3 player or both of them.

Feature models are used in different scenarios of software production ranging from model driven development feature oriented programming soft-ware factories or generative programming, all of them around software product line development. Although feature models are studied in software product line engineering, these information models can be used in different contexts ranging from requirements gathering to data model structures, hence the potential importance of feature models in the information systems domain.

The term *feature model* was coined by Kang et al. in the FODA report back in 1990 and has been one of the main topics of research in software products [4].

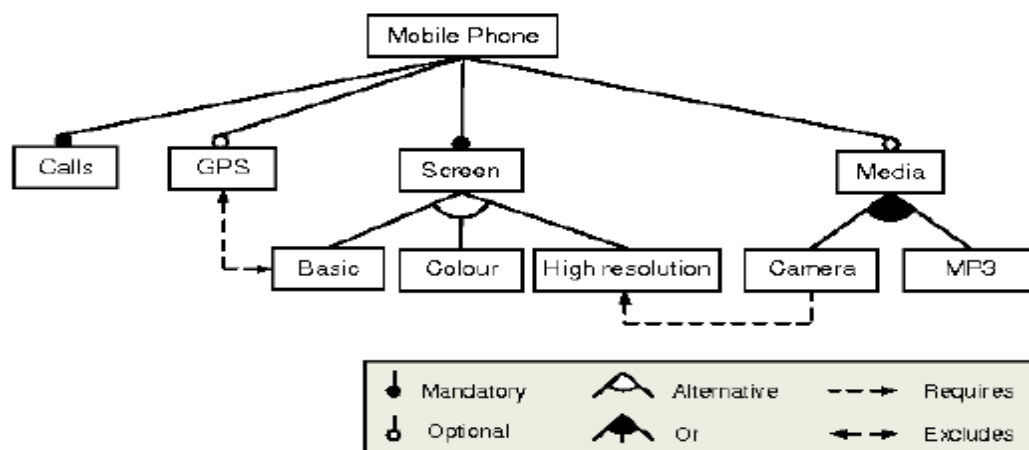


Figure 3.1: A sample feature model

There are different feature model languages. We review the most well-known notations for those languages.

3.1.1 Basic feature models

We group as basic feature models those allowing the following relationships among features:

- **Mandatory.** A child feature has a mandatory relationships with its parent when the child is included in all products in which its parent feature appears. For instance, every mobile phone system in our example must provide support for *calls*.

- **Optional.** A child feature has an optional relationship with its parent when the child can be optionally included in all products in which its parent feature appears. In the example, software for mobile phones may optionally include support for GPS.
- **Alternative.** A set of child features have an alternative relationship with their parent when only one feature of the children can be selected when its parent feature is part of the product. In the example, mobile phones may include support for a basic, color or high resolution screen but only one of them.
- **Or.** A set of child features have an or-relationship with their parent when one or more of them can be included in the products in which its parent feature appears. In Figure 1, whenever Media is selected, Camera, MP3 or both can be selected.

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

- **Requires.** If a feature A requires a feature B, the inclusion of A in a product implies the inclusion of B in such product. Mobile phones including a camera must include support for a high resolution screen.
- **Excludes.** If a feature A excludes a feature B, both features cannot be part of the same product. GPS and basic screen are incompatible features.

More complex cross-tree relationships have been proposed later in the literature allowing constraints in the form of generic propositional formulas, e.g. “A and B implies not C”.

3.1.2 Cardinality-based feature models

Some authors propose extending FODA feature models with UML-like multiplicities (so-called *cardinalities*). Their main motivation was driven by practical applications and

“conceptual completeness”. The new relationships introduced in this notation are defined as follows:

- **Feature cardinality:** A feature cardinality is an sequence of intervals denoted $[n, m]$ with n as lower bound and m as upper bound. These intervals determine the number of instances of the feature that can be part of a product. This relationship may be used as a generalization of the original mandatory $([1,1])$ and optional $([0,1])$ relationship in FODA
- **Group cardinality:** A group cardinality is an in-terval denoted $hn..mi$, with n as lower bound and m as upper bound limiting the number of childfeatures that can be part of a product.[5].

3.1.3 Extended feature models

Sometimes it is necessary to extend feature models to include more information about features. This information is added in terms of so-called feature at-tributes. This type of models where additional information is included are called extended, advanced or attributed feature models.

FODA, the seminal report on feature models, already contemplated the inclusion of some additional information in feature models. For instance, relation-ships between features and feature attributes were introduced. Later, Kang et al. make an explicit reference to what they call “non-functional” features related to feature attributes. In addition, other groups of authors have also proposed the inclusion of attributes in feature models There is no consensus on a notation to define attributes. However, most proposals agree that an attribute should consist at least of a *name*, a *domain* and a *value*. Fig-ure 2 depicts a sample feature model including at-tributes using the notation proposed by Benavides. As illustrated, attributes can be used to specify extra-functional information such as cost, speed or RAM memory required to support the feature. [7]

Extended feature models can also include complex constraints among attributes and features like: “ If at-tribute A of feature F is lower than a value X, then feature T cannot be part of the product”.

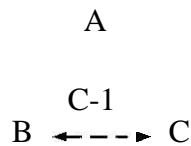


Figure 3.2: A void feature model

as “model validation” , “model consistency checking” , “model satisfiability checking” , “model solve-ability checking“ and “model constraints checking”.

3.2 Analysis operations on feature models

In this section, we answer RQ1: What operations of analysis on feature models have been proposed? For each operation, its definition, an example and possible practical applications are presented [9].

3.2.1 Void feature model

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

As an example, Figure 3.2 depicts a void feature model. Constraint C-1 makes the selection of the mandatory features B and C not possible, adding a contradiction to the model because both features are mandatory.

3.2.2 Valid product

This operation takes a feature model and a product (i.e. set of features) as input and returns a value that determines whether the product belongs to the set of products represented by the feature model or not. For instance, consider the products P1 and P2, described below, and the feature model.

$P1 = \{ \text{MobilePhone, Screen, Colour, Media, MP3} \}$

$P2 = \{ \text{MobilePhone, Calls, Screen, Highresolution, GPS} \}$

Product P1 is not valid since it does not include the mandatory feature Calls. On the other hand, product P2 does belong to the set of products represented by the model.

This operation may be helpful for software product line analysts and managers to determine whether a given product is available in a software product line. This operation is sometimes also referred to as

“valid configuration checking”, “valid single system”, “configuration consistency”, “feature compatibility”, “product checking” and “product specification completeness” .

3.2.3 Valid partial configuration

This operation takes a feature model and a partial configuration as input and returns a value information Whether, the configuration is valid or not, i.e. a partial configuration is valid if it does not include any contradiction. Consider as an example the partial configurations C1 and C2, described below, and the feature model of Figure 3.1.

$C1 = (\{ \text{MobilePhone, Calls, Camera} \}, \{ \text{GPS, High resolution} \})$

$C2 = (\{ \text{MobilePhone, Calls, Camera} \}, \{ \text{GPS} \})$

C1 is not a valid partial configuration since it selects support for the camera and removes the high resolution screen that is explicitly required by the software product line. C2 does not include any contradiction and therefore could still be extended to a valid full configuration.

This operation results helpful during the product derivation stage to give the user an idea about the progress of the configuration. A tool implementing this operation could inform the user as soon as a configuration becomes invalid, thus saving time and effort.

3.2.4 All products

This operation takes a feature model as input and returns all the products represented by the model. For instance, the set of all the products of the feature model presented in Figure 3.1 is detailed below:

P1 = { MobilePhone, Calls, Screen, Basic }
P2 = { MobilePhone, Calls, Screen, Basic, Media, MP3 }
P3 = { MobilePhone, Calls, Screen, Colour }
P4 = { MobilePhone, Calls, Screen, Colour, GPS }
P5 = { MobilePhone, Calls, Screen, Colour, Media, MP3 }
P6 = { MobilePhone, Calls, Screen, Colour, Media, MP3, GPS }
P7 = { MobilePhone, Calls, Screen, High resolution }
P8 = { MobilePhone, Calls, Screen, High resolution, Media, MP3 }
P9 = { MobilePhone, Calls, Screen, High resolution, Media, MP3, Camera }
P10 = { MobilePhone, Calls, Screen, Highresolution, Media, Camera }
P11 = { MobilePhone, Calls, Screen, Highresolution, GPS }

P12 = { MobilePhone, Calls, Screen, High resolution, Media, MP3, GPS }
P13 = { MobilePhone, Calls, Screen, Highresolution, Media, Camera, GPS }
P14 = { MobilePhone, Calls, Screen, High resolution, Media, Camera, MP3, GPS }

This operation may be helpful to identify new valid requirement combinations not considered in the initial scope of the product line. The set of products of a feature model is also referred to in the literature as “allvalid configurations” and “list of products” [7].

3.2.5 Number of products

This operation returns the number of products represented by the feature model received as input. Note that a feature model is void iff the number of products represented by the model is zero.

This operation provides information about the flexibility and complexity of the software product line. A big number of potential products may reveal a more flexible as well as more complex product line. The number of products of a feature model is also referred to in the literature as “variation degree”.

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

For instance, the set of products of the feature model in Figure 1 applying the partial configuration $(S, R) = (\{Calls, GPS\}, \{Colour, Camera\})$, being S the set of features to be selected and R the set of features to be removed, is:

$P1 = \{MobilePhone, Calls, Screen, Highresolution, GPS\}$

$P2 = \{MobilePhone, Calls, Screen, High resolution, Media, MP3, GPS\}$

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

3.2.6 Anomalies detection

A number of analysis operations address the detection of anomalies in feature models i.e. undesirable properties such as redundant or contradictory information. These operations take a feature model as input and return information about the anomalies detected. We identified five main types of anomalies in feature models reported in the literature. These are

Dead features: A feature is dead if it cannot appear in any of the products of the software product line. Dead features are caused by a wrong usage of cross-tree constraints. These are clearly undesired since they give the user a wrong idea of the domain.

False optional features: A feature is false optional if it is included in all the products of the product line despite not being modeled as mandatory. Figure 3.1 depicts some examples of false optional feature.

Conditionally dead features: A feature is condition-ally dead if it becomes dead under certain circumstances (e.g. when selecting another feature). Both unconditional and

conditional dead features are often referred to in the literature as “contradictions” or “inconsistencies”. In Figure 3.2 feature B becomes dead whenever feature D is selected. Note that, with this definition, features in an alternative relationship are conditionally dead.

Redundancies: A feature model contains redundancies when some semantic information is modeled in multiple ways. Generally, this is regarded as a negative aspect since it may decrease the maintainability of the model. Nevertheless, it may also be used as a means of improving readability and comprehensibility of the model. Figure 3 depicts some examples of redundant constraints in feature models [8].

3.2.7 Explanations

This operation takes a feature model and an analysis operation as inputs and returns information (so-called explanations) about the reasons of *why* or *why not* the corresponding response of the operation. Causes are mainly described in terms of features and/or relationships involved in the operation and explanations are often related to anomalies.

Presents a feature model with a dead feature. A possible explanation for the problem would be “Feature D is dead because of the excludes constraint with feature B”. We refer the reader to a detailed analysis of explanation operation.

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

3.2.9 Corrective explanations

This operation takes a feature model and an analysis operation as inputs and returns a set of corrective explanations indicating changes to be made in the original inputs in order to change the output of the operation. In general, a corrective explanation provides suggestions to solve a problem, usually once this has been detected and explained.

For instance, some possible corrective explanations to remove the dead feature in Figure 7 would be “remove excludes constraint C-1” or “model feature B as optional”. This operation is also referred to in the literature as “corrections”.

3.2.10 Feature model relations

This operation takes two different feature models as inputs and returns a value informing how the models are related. The set of features in both models are not necessarily the same. These operations are useful for determining how a model has evolved over time.

Refactoring: A feature model is a refactoring of another one if they represent the same set of products while having a different structure. Refactoring is useful to restructure a feature model without changing its semantics. When this property is fulfilled the models are often referred to as “equivalent”.

Arbitrary edit: There is no explicit relationship between the input models, i.e. there are none of the relationships defined above.

3.2.11 Optimization

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

This operation is chiefly useful when dealing with extended feature models where attributes are added to features. In this context, optimization operations may be used to select a set of features maximizing or minimizing the value of a given feature attribute [8].

3.2.12 Core features

This operation takes a feature model as input and returns the set of features that are part of all the products in the software product line. For instance, the set of core features of the model presented in Figure 3.1 is

{MobilePhone,Calls,Screen}.

Core features are the most relevant features of the software product line since they are supposed to appear in all products. Hence, this operation is useful to determine which features should be developed in first place or to decide which features should be part of the core architecture of the software product line.

3.2.13 Variant features

This operation takes a feature model as input and returns the set of variant features in the model. Variant features are those that do not appear in all the products of the software product line. For instance, the set of variant features of the feature model presented in Figure 1 is {Basic,Colour,Highresolution,Media,Camera, MP3,GPS}.

3.2.14 Dependency analysis

This operation takes a feature model and a partial configuration as input and returns a new configuration with the features that should be selected and/or removed as a result of the propagation of constraints in the model. As an example, consider the input and output configurations described below and the model in Figure 3.1.

Input = ({MobilePhone,Calls,Camera}, {MP3})

Output = ({MobilePhone,Calls,Camera,Media,Screen,High resolution},

{MP3,Basic,Color})

Features Screen and High resolution are added to the configuration to satisfy the requires constraint with Camera. Media is also included to satisfy the parental relationship with

Camera. Similarly, features Basic and Color are removed to fulfill the constraints imposed by the alternative relationship.

This operation is the basis for constraint propagation during the interactive configuration of feature models.

Chapter 4

Analysis of Feature Model Using Miranda

4.1 Software Product Line

In software development, a **feature model** is a compact representation of all the products of the Software Product Line (SPL) in terms of "features". Feature models are visually represented by means of feature diagrams. Feature models are widely used during the whole product line development process and are commonly used as input to produce other assets such as documents, architecture definition, or pieces of code.

A SPL is a family of related programs. When the units of program construction are features—increments in program functionality or development—every program in an SPL is identified by a unique and legal combination of features, and vice versa.

A "feature" is defined as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system". The focus of SPL development is on the systematic and efficient creation of similar programs. FODA is an analysis devoted to identification of features in a domain to be covered by a particular SPL [4].

The feature models we consider in this paper consist of a feature tree and a set of constraints. A feature tree is a tree which can have three kinds of nodes: MandOpt nodes, Or nodes and Xor nodes. A MandOpt node has two sets of child nodes, called mandatory and optional nodes respectively. Or nodes and Xor nodes have 2 or more child nodes. A leaf of the tree is a MandOpt node without children. Just for the ease

of writing concise algorithms, we assume the existence of a special feature tree NIL, which has no nodes. Each node of a tree has a feature, which is just a list of characters. All nodes in a feature tree have different features, and NIL does not occur as sub tree of any feature tree. A product is a set of features. A constraint maps products to Boolean values; in our prototype implementation the constraints are restricted to constraints of the forms "A requires B" and "A excludes B".

4.2 Model

A feature model is a model that defines features and their dependencies, typically in the form of a feature diagram + left-over (a.k.a. cross-tree) constraints. But also it could be as a table of possible combinations.

4.3 Diagram

A feature diagram is a visual notation of a feature model, which is basically an and-or tree. Other extensions exist: cardinalities, feature cloning, feature attributes, discussed below.

4.4 Configuration

A feature configuration is a set of features which describes a member of an SPL: the member contains a feature if and only if the feature is in its configuration. A feature configuration is permitted by a feature model if and only if it does not violate constraints imposed by the model.

4.5 Feature Model Notation

Current feature modeling notations [6] may be divided into three main groups, namely:

- Basic feature models
- Cardinality-based feature models
- Extended feature models

4.6 Basic feature models

Relationships between a parent feature and its child features (or sub features) are categorized as:

Mandatory: A child feature is said to be mandatory when it is required to appear when the parent feature appears. For instance, it is mandatory to have a special platform for Android mobile phone.

Optional: A child feature is said to be optional when it can or not appear when the parent features appears. For instance, it is optional to have a pdf reader software in the mobile phone.

Alternative: A set of child features are said to be alternative when only one child feature can be selected when the parent feature appears. For instance a Gamer cannot select both Automatic and Manual for car control during car selection.

Optional Alternative: One feature from a set of alternative features may or may not be included if parent included.

Or: A set of child features are said to have an or-relation with their parent when one or more sub features can be selected when the parent feature appears. For instance, the engine of a car can be electric, gasoline or both at the same time.

Optional Or: one or more optional feature may be included if the parent is included.

A feature model can be considered as a graph consists of a set of sub graphs. Each sub graph is created separately by defining a relationship between the variation point (denoted as v_i) and the variants $(v_{i,j})$ by using the expressions shown in Fig.4.1. The complexity of a graph construction lies in the definition of dependencies among variants. When there is a relationship between cross-tree (or cross hierarchy) variants (or variation points) we denote it as a dependency. Typically dependencies are either *inclusion* or *exclusion*: if there is a dependency between p and q, then if p is included then q must be included (or excluded). Dependencies are drawn by dotted lines.




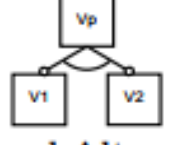

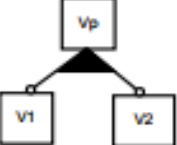
Type	Logic Expression	Type	Logic Expression
 Mandatory	$v_p \Leftrightarrow v$	 Optional	$v \Rightarrow v_p$
 Alternative	$v_p \Leftrightarrow (v_1 \oplus v_2)$	 Optional Alternative	$(v_1 \oplus v_2) \Rightarrow v_p$
 Or	$v_p \Leftrightarrow (v_1 \vee v_2)$	 Optional Or	$(v_1 \vee v_2) \Rightarrow v_p$

Figure 4.1: Logical notations for feature model

The Feature model of the CAD system is splitted into smaller part for the convenience of analysis. Then we analysis each part individually and get some basic rules [5].

4.7 Scenario of Feature Models

4.7.1 Scenario 1

In Fig 4.2, v1 and v2 are variants (and variation points) and there is a require

dependency between them. Here v2 is selected whenever v1 is selected.

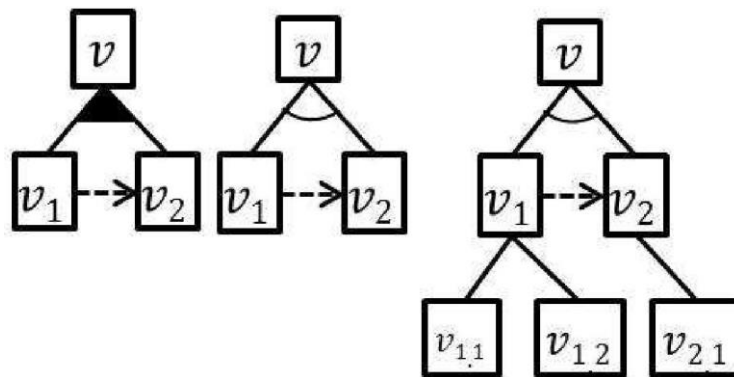


Figure 4.2: Require dependency between variants and between variation points

Adopting the notation in [13] we define the following rule for dependency among variants as well as variation points.

$\forall v_1, v_2 \bullet \text{type}(v_1, \text{variant}) \wedge \text{type}(v_2, \text{variant}) \wedge$

$\text{require } v \ v(v_1, v_2) \wedge \text{select}(v_1) = \text{select}(v_2)$

$\forall v_1, v_2 \bullet \text{type}(v_1, \text{variation point}) \wedge \text{type}(v_2, \text{variation point})$

$\wedge \text{require } vp \ vp(v_1, v_2) \wedge \text{select}(v_1) = \text{select}(v_2)$

where $\text{type}(v, \dots)$ indicates whether v is a variant or variation point, $\text{select}(v)$ indicates the selection of variant v and $\text{require}()$ indicates the require relationship. Similar notation will be used for rest of the rules definition. Due to this dependency rule the dependant variant, v_2 here, will always be selected if v_1 is selected and such selection will not be affected by the type of relationship such as Alternative, with their parent.

Scenario 2

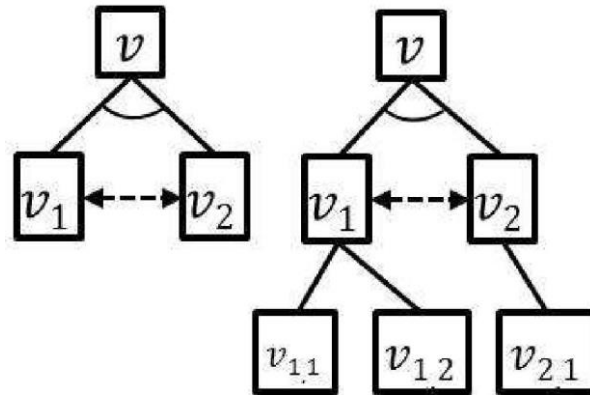


Figure 4.3: Exclude dependency between variants and between variation points

In Fig 4.3, there is an exclude relationship between v_1 and v_2 . Here v_1 and v_2 are variants in the left figure and variation point in the right part of the figure. In both cases, as there is exclude relationship between them only one can be selected at a time. Here, we can suggest that for such scenario the relationship among the variants or variation points must be Alternative to keep the feature mode well-formed. Similar to previous example, we define rules for such dependencies.

$$\forall v_1, v_2 \cdot \text{type}(v_1, \text{variant}) \wedge \text{type}(v_2, \text{variant}) \wedge$$

$$\text{exclude } v \ v(v_1, v_2) \wedge \text{select}(v_1) \Rightarrow \text{notselect}(v_2)$$

$$\forall v_1, v_2 \cdot \text{type}(v_1, \text{variation point}) \wedge$$

$$\text{type}(v_2, \text{variation point}) \wedge \text{exclude } vp \ vp(v_1, v_2) \wedge$$

$$\text{select}(v_1) \Rightarrow \text{notselect}(v_2)$$

4.7.3 Scenario 3

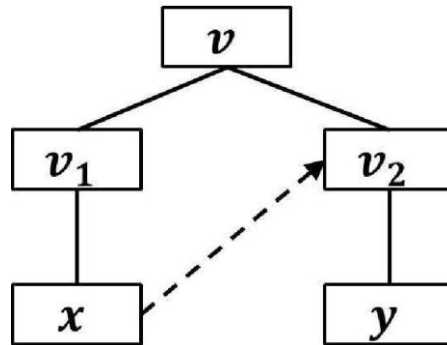


Figure 4.4: Dependency between variants and variation point

In Fig 4.4, Suppose v_1 and v_2 are two variation points. x is a variant under the variation point v_1 and y is a variant under the variation point v_2 . There is a require relationship between the variant x and the variation point v_2 . That means when we select x , v_2 will be automatically selected.

From this scenario we can derive a rule

$$\forall v_1, v_2, x, y \cdot \text{type}(x, \text{variant}) \wedge \text{type}(v_1, \text{variation point}) \\ \wedge \text{type}(v_2, \text{variation point}) \wedge \text{requires } v \text{ vp}(x, v_2) \wedge \text{select}(x) \Rightarrow \text{select}(v_2)$$

4.7.4 Scenario 4

In Fig 4.5, Suppose v_1 and v_2 are two variation points. x is a variant under the variation point v_1 , and y is a variant of the variation point v_2 . There exists an exclude relationship between the variant x and the variation point v_2 . That means when we select x , v_2 will be automatically deselected because the selection of the variant x cannot allow the selection of the variation point v_2 . That means both the variation point v_1 and v_2 cannot appear in a product.

In Fig 5.6, Suppose v_1 and v_2 are two variation points. x is a variant under the variation point v_1 , and y is a variant of the variation point v_2 . There exists an exclude relationship between the variant x and the variation point v_2 . That means when we select x , v_2 will be automatically deselected because the selection

of the variant x cannot allow the selection of the variation point v_2 . That means both the variation point v_1 and v_2 cannot appear in a product.

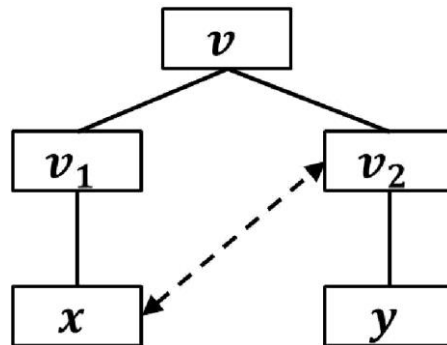


FIGURE 4.5: Exclude dependency between variants and variation point

The following rule is derived from this scenario

$\mathbf{V} v_1, v_2, x, y$ type(x , variant) \mathbf{A} type(v_1 , variation point)

\mathbf{A} type(v_2 , variation point) \mathbf{A} exclude v vp(x, v_2) \mathbf{A} select(x) = notselect(v_2)

For all variants x and variation point v_2 ; if x excludes v_2 and x is selected, then x_2 should not be selected.

4.7.5 Scenario 5

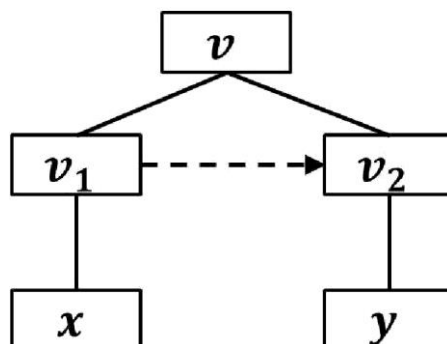


Figure 4.6: Requires dependency between variation points

In Fig. 4.6, v_1 and v_2 are two variation point and x and y are their variants respectively. There is a requires relationship between the variation point v_1 and v_2 , then when the variation point v_1 is selected we must select the variation point v_2 , otherwise the condition will be violated. In other words, the selection of variation point v_1 will automatically select the variation point v_2 .

From this analysis we can derive a rule that can satisfy when this type of scenario occurs in the feature model.

4.7.6 Scenario 6

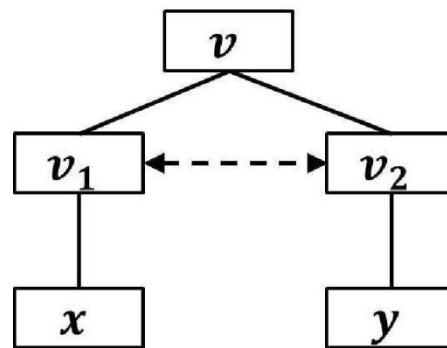


Figure 4.7: Exclude dependency between variation points

In Fig. 4.7, suppose v_1 and v_2 are two variation points. Let there exists is an exclude relationship between the variation points v_1 and v_2 . Hence when variation point v_1 is selected we must deselect the variation point v_2 . In other way we can say that selection of variation point v_1 will automatically reject the selection of variation point v_2 . From this analysis we can derive a rule that can satisfy when this type of scenario occur in the feature model [7].

4.7.7 Scenario 7

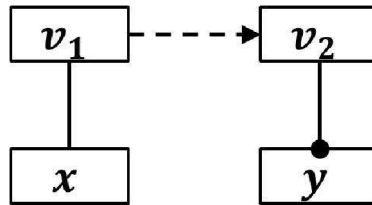


Figure 4.8: Variant and variation point relation

In Fig. 4.8, v_1 is variation point and x is a variant of that variation point. When a variation is selected its variation point will be selected automatically. This scenario can also be called as parent-child, when a child is selected, its parent will be selected as well. The following rule is defined for this scenario.

V v_1, v_2 type(x , variant) **A** type(v_1 , variation point)

A variant(v_1, x) **A** select(x) = select(v_1)

4.7.8 Scenario 8

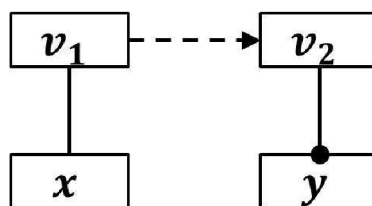


Figure 4.9: Variation point to variation point and parent-child relation

Suppose v_1 and v_2 are two variation point and x and y are their respective variants and there is a requires relation from v_1 to v_2 . Here y is a mandatory feature. In this case when variant x is selected according to our earlier scenario, variant y will

V v_1, v_2, x, y type(x , variant) **A**type(y , variant) **A**variants(v_1, x) **A**variant(v_2, y)
A common(y) **A** requires vp vp(v_1, v_2) **A** select(x) = select(y)

also be selected. Fig.4.9 shows the scenario and the corresponding definition of rules is as follows:

4.7.9 Scenario 9

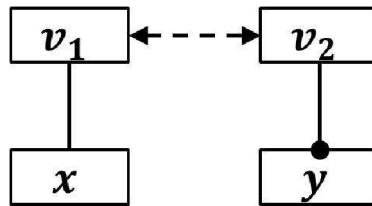


Figure 4.10: Variant and variation point exclude relation

Suppose v_1 and v_2 are variation points. x and y are two of their variants respectively. There exists an exclude relationship between v_1 and the variants v_2 . If someone selects the variants x it will automatically deselect the selection of the variant y , because we cannot select a variant when its variation point is not selected. The scenario is depicted in Fig 4.10

V v_1, v_2, x, y : type(x , variant) **A** type(y , variant) **A** variant(v_1, x) **A** variant(v_2, y)
A common(y, y) **A** requires vp vp(v_1, v_2) **A** select(x) = not select(y)

4.8 Feature models

The feature models we consider in this paper consist of a feature tree and a set of constraints. A feature tree is a tree which can have three kinds of nodes: MandOpt nodes, Or nodes and Xor nodes. A MandOpt node has two sets of child nodes, called mandatory and optional nodes respectively. Or nodes and Xor nodes have 2 or more child nodes. A leaf of the tree is a MandOpt node without children. Just for the ease of writing concise algorithms, we assume the existence of a special feature tree NIL, which has no nodes. Each node of a tree has a feature, which is just a list of characters. All nodes in a feature tree have different features, and NIL does not occur as subtree of any feature tree. A product is a set of features. A constraint maps products to Boolean values; in our prototype implementation the constraints are restricted to constraints of the forms "A requires B" and "A excludes B".

In Miranda, these type definitions are as follows:

```
tree ::= MandOpt feature [tree] [tree] | Or feature [tree] |
      Xor feature [tree] |
      NIL
feature == [char]
product == [feature]
constraint ::= Requires feature feature | Excludes feature
            feature feature_model == (tree, [constraint])
```

4.9 Generalised feature trees

Features in a feature tree are, albeit implicitly, required to be all distinct. In the generalisation of feature trees we consider in this paper, this requirement is somewhat

relaxed. We define a generalised feature tree (GFT) to be a feature tree whose features, instead of being required to be all distinct, satisfy the following two restrictions:

- Restriction 1: when two nodes of a GFT have the same feature, they belong to different subtrees of an Xor node.
- Restriction 2: for each node of a GFT, all subtrees have disjoint semantics.

In we gave an implementation in Miranda of functions with type definitions

```
commit :: feature -> tree -> tree
delete :: feature -> tree -> tree
```

The resulting trees are indeed GFTs. Restriction 1 is satisfied because all generated subtrees have the same Xor root node as parent node. Restriction 2 is satisfied because the semantics of the generated subtrees are the parts of a partition of P, and therefore have no common features.

In [6] we gave an implementation in Miranda of a function with type definition

```
elimConstr :: feature_model -> tree
```

The argument of this function is a feature model which consists of a feature tree and constraints of the forms "A requires B" and "A excludes B"; the function returns a corresponding GFT.[7]

4.10 Analysis of feature models

In this section we show how the algorithms of the preceding section can be used to analyze feature models which consist of a feature tree and a number of constraints. In this implementation the constraints are restricted to be of the forms the function `elimConstr` can handle; these are the forms "A requires B" and "A excludes B", but other forms might be included as well, as described in the previous section.

Starting point of the analysis is a feature model consisting of the feature tree `f_tree` and the list of constraints `constraints`. The first step of the analysis is the computation of an equivalent GFT `gft`:

```
gft :: tree
gft = elimConstr (f_tree, constraints)
```

The function `elimConstr` here can be used if all constraints are of the forms "A requires B" and "A excludes B"; otherwise, the procedure described in the previous section should be followed.

In the remainder of this section we describe the computation of a number of properties of the specified software product line [4].

4.10.1 Existence of products

The feature model has products if and only if `gft` is not equal to `NIL`:

```
has_products :: bool
has_products = gft /= NIL
```

4.10.2 Dead features

The dead features of the feature model are the features which occur in features but do not occur in `gft`:

```
dead_features :: [feature]
dead_features
  = features f_tree -- features gft
```

Here, the function `features` computes a list of all features of a GFT:

```
features :: tree -> [feature]
features (MandOpt f ms
os)
  = f : concat (map features (ms++os)) features (Or f fts)
  = f : concat (map features fts) features (Xor f fts)
  = f : concat (map features fts)
```

4.10.3 Number of products

The number of products of the feature model is

```
nr_products :: num
nr_products = nrProds gft
```

where the function nrProds is given by

```
nrProds :: tree -> num
nrProds NIL = 0
nrProds (MandOpt nm ms os)
= product (map nrProds ms) *
  product (map (+1) (map nrProds os)) nrProds (Xor nm fts)
= sum (map nrProds fts)
nrProds (Or nm fts)
= product (map(+1) (map nrProds fts)) - 1
```

4.10.4 List of all products

A list of all products of the feature model is

```
list_of_products :: [[feature]]
list_of_products = products gft
```

where the function products computes a list of products of a GFT:

```
products :: tree -> [[feature]]
products (MandOpt x ms os)
= map(x:) (f (map products ms ++
map([]:) (map products os)))
  where
  f [] = [[]]
```



```

f(xs:xss) = [u++v|u<-xs;v<-f xss] products (Xor x fts)
= map(x:) (foldl(++)[](map products fts)) products (Or x fts)
= map(x:) (f(map products fts)--[[]]) where
f [] = [[]]
f(xs:xss) = [u++v|u<-([]:xs);v<-f xss]

```

4.10.5 Products which contain a given set of features

A GFT whose products are precisely those products of gft which contain all features from a list `required_features` is:

```

gft2 :: tree
gft2 = gft_req_fts required_features gft
where the function gft_req_fts is defined by:
gft_req_fts :: [feature] -> tree -> tree
gft_req_fts [] t = t

gft_req_fts (f:fs) t
    = commit f (gft_req_fts fs t)

```

4.11 Minimal set of conflicting constraints

A set of constraints is in conflict with a feature tree if the feature model consisting of this tree and these constraints has no products, i.e when gft evaluates to NIL. A user, confronted with such a conflict, may want some explanation of this. A solution might be to provide the user with a smallest minimal set of constraints that conflict with the feature tree. A minimal set of constraints is a set which contains conflicting constraints, but has no proper subset whose constraints also conflict. A smallest minimal set of conflicting constraints can be computed by

```

confl_constr :: [constraint]
confl_constr = smsocc(f_tree,constraints)
where the function smsocc (smallest minimal set of
conflicting constraints) is given by:
smsocc :: feature_model -> [constraint] smsocc (t,[]) = []
smsocc (t,c:cs)
  = [c], if t2 = NIL
  = [], if set1 = []
  = c:set1, if set2 = [] \ / #set2>#set1 = set2, otherwise
  where
    t2 = elimConstr (t,[c])
    set1 = smsocc (t2,cs)
    set2 = smsocc (t,cs)

```

This function, given the original feature model as argument, returns a list with a minimal set of conflicting constraints if gft equals NIL; otherwise it returns the empty list.

4.12 Explanation of dead feature

If `dead_features`, the list of dead features, is non-empty and contains the feature `dead_feature`, the user might want explanation why this feature is dead. As above, this explanation is a minimal set of constraints which causes the feature to be dead. It is given by

```

expl_dead_ft :: [constraint] expl_dead_ft
  = explain (f_tree,constraints) dead_feature

```

where the function `explain` is given by

```

explain :: feature_model
        -> feature -> [constraint] explain (t,cs) f
= smsocc (t2,cs), if t2 /= NIL = [], otherwise

where
t2 = commit f t

```

The arguments of this function are the original feature model and a feature from the list dead features. It returns a minimal set of constraints which causes the feature to be dead. If the feature does not belong to dead features, the empty list is returned [13].

4.13 Example of Feature Tree using Miranda

As an example, consider the feature tree T in Figure 4.11,

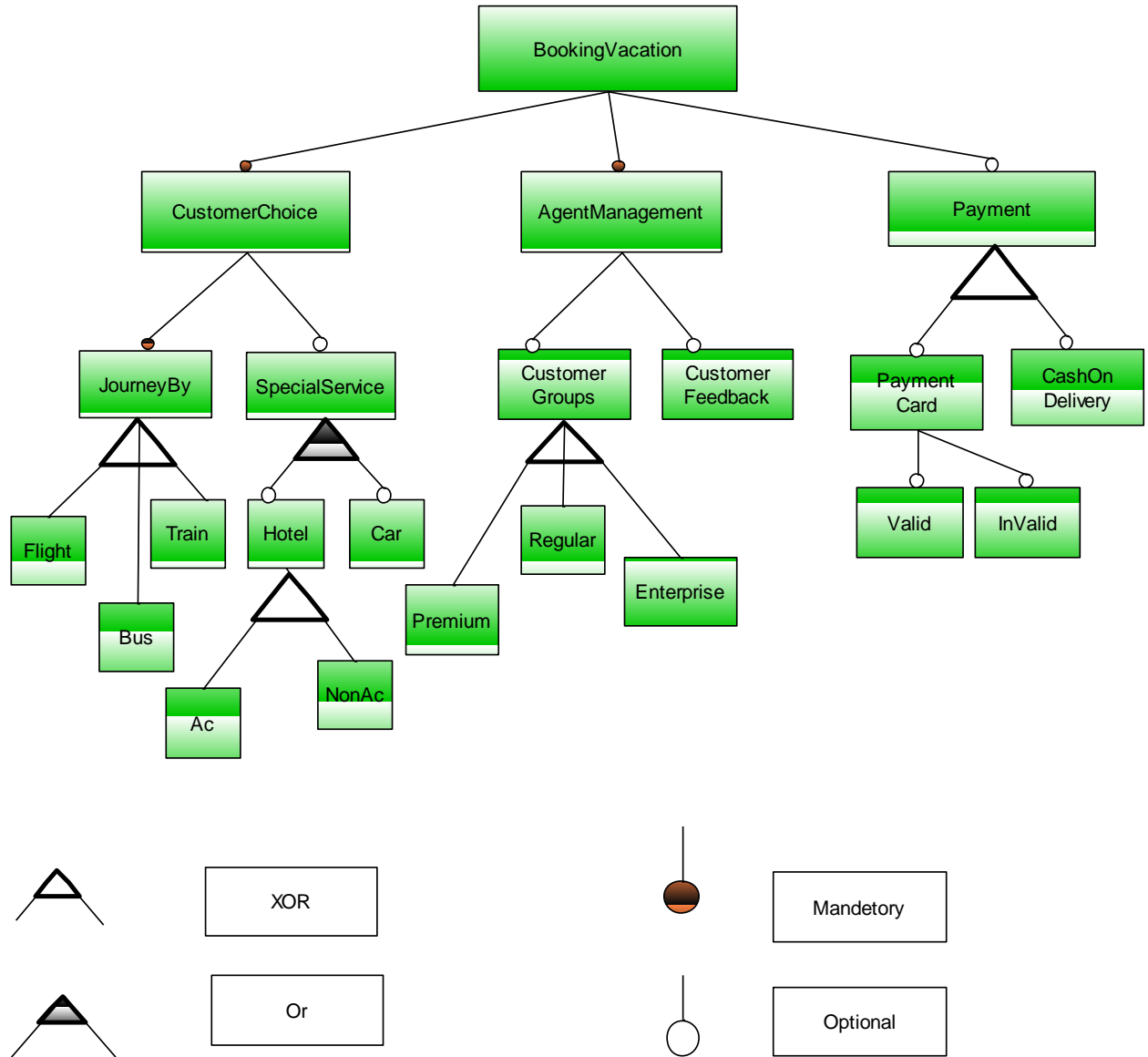


Figure 4.11: Example feature tree T in Miranda

4.13.1 The definition of f_tree is:

1. f_tree = MandOpt "BookingVacation" [n1,n2][n3]
2. n1 = MandOpt "CustomerChoice" [n4][n5]
3. n2 = MandOpt "AgentManagement" [n7][n6]
4. n3 = Xor "Payment" [n8,n9]
5. n4 = Xor "JourneyBy" [n10,n11,n12]
6. n5 = Or "SpecialService" [n13,n14]
7. n6 = Xor "CustomerGroups" [n15,n16,n17]
8. n7 = MandOpt "CustomerFeedback" [] []
9. n8 = MandOpt "paymentCard" [] [n18,n19]
10. n9 = MandOpt "CashOnDelivery" [] []
11. n10 = MandOpt "Flight" [] []
12. n11 = MandOpt "Bus" [] []
13. n12 = MandOpt "Train" [] []
14. n13 = MandOpt "Hotel" [] [n20,n21]
15. n14 = MandOpt "Car" [] []
16. n15 = MandOpt "Premium" [] []
17. n16= MandOpt "Regular" [] []
18. n17 = MandOpt "Enterprise" [] []
19. n18 = MandOpt "Valid" [] [n22]
20. n19 = MandOpt "Invalid" [] []
21. n20 = MandOpt "Ac" [] []
22. n21 = MandOpt "NonAc" [] []

4.14 Constraints

Our example feature model consists of this feature tree T and the 2 constraints: "CashOnDelivery excludes Consumer" and "Enterprise requires Consumer". So the list constraints is given by

```
constraints = [c1,c2]
```

```
c1 = Excludes "CashOnDelivery" "Consumer"
```

```
c2 = Requires "Enterprise" "Consumer"
```

The GFT gft is given. It could have been computed with the function `elimConstr`, since both constraints are of the forms "A requires B" and "A excludes B"; however, we will illustrate the method to derive it which was given in section 4. If "Consumer" is present in a product, the constraints are satisfied iff "CashOnDelivery" is not present. If "Consumer" is absent in a product, the constraints are satisfied iff "Enterprise" is also absent. So the set of products P can be partitioned in such a way that the parts $P(+Consumer-CashOnDelivery)$ and $P(-Consumer-Enterprise)$ consist of the products which satisfy the constraints. Therefore, the equivalent GFT is given by a new Xor node which has $T(+Consumer-CashOnDelivery)$ and $T(-Consumer-Enterprise)$ as subtrees.

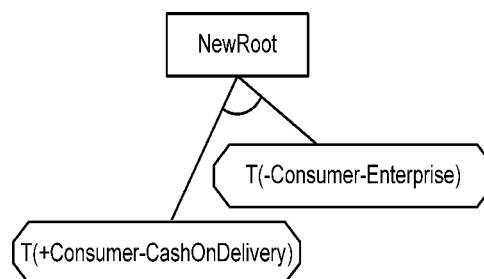


Figure 4.12: generalised feature tree gft , top-level

The analysis of this example feature model proceeds as follows:

- `has_products` evaluates to True.
- `dead_features` evaluates to ["Enterprise"], showing that the feature "Enterprise" is dead.
- `nr_products` evaluates to 40.
- `list_of_products` evaluates to a list of the 40 products (not shown for brevity).
- if `required_features` is defined as a list of features then `gft2` evaluates to a GFT which can be analyzed in the same manner.
- if `dead_feature` is defined to be "Enterprise" then `expl_dead_ft` evaluates to [Requires "Enterprise" "Consumer"] showing that the second constraint is on its own responsible for the deadness of "Enterprise".

Now, suppose that an extra constraint "SalesProcessing requires Enterprise" is added:

```
constraints = [c1, c2, c3]
c3 = Requires "SalesProcessing"
      "Enterprise"
```

Now `has_products` evaluates to False and `confl_constr` evaluates to [Requires "Enterprise" "Consumer", Requires "Sales Processing" "Enterprise"] which shows that the second and third constraints together form a smallest minimal set of constraints that conflict with the feature tree.

4.15 Computational Complexity

We have shown in that the decision problem whether a feature model which is given by a feature tree and a set of constraints is NP-complete. Therefore, we cannot hope that the analysis

of such a feature model can be performed in polynomial time in the worst case. Indeed, the construction of the GFT for the feature model takes a time which is exponential in the number of constraints in the worst case. Also the algorithm for the computation of a minimal set of constraints which conflict with the feature tree and the algorithm which computes the minimal set of constraints which cause a feature to be dead are exponential in the number of constraints. However, once the GFT has been constructed, the algorithms for the existence of products, the number of products and the list of dead features are linear in the size of the GFT .

We have introduced the concept of generalised feature trees and have shown how they can be used to analyse feature models which consist of a feature trees and additional constraints. Detailed algorithms have been given in the functional programming language Miranda. The algorithms are efficient when the constraints are a small number of "requires" and/or "excludes" constraints [14].

Chapter 5

Summary and Future Work

5.1 Summary

We have introduced the concept of Software Product Line (SPL) Feature Tree, Logical Notation of Feature Tree, Miranda Language and It's Function, Representation of Feature tree with Miranda, Verification of Dead Features, False Optional Determination of List of Products .

Also, we introduced here generalised feature trees and have shown how they can be used to analyse feature models which consist of a feature trees and additional constraints. Detailed algorithms have been given in the functional programming language Miranda. The algorithms are efficient when the constraints are a small number of "requires" and/or "excludes" constraints.

5.2 Future Work

In future we want to work with this project. We want to improve this system. There are some features we want to add though the system, they are given below

- We have not verified all constraints yet. In future, we will verify all constraints.
- In future we will work with generalized feature tree in Miranda Environment and also we will try to run the code in different functional language.

Bibliography

1. Analysis of Feature Models using Generalised Feature Trees- Pim van den Broek and Ismênia Galvão Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands. Conference: Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings
2. D. Batory, "Feature Models, Grammars, and Propositional Formulas", in: H. Obbink and K. Pohl (eds.): Software Product Lines Conference 2005, Lecture Notes in Computer Science 3714, Springer-Verlag Berlin Heidelberg, pp. 7-20, 2005.
3. P. van den Broek, I. Galvão and J. Noppen, "Elimination of Constraints from Feature Trees" in: Steffen Thiel and Klaus Pohl (Eds.), Proceedings of the 12th International Software Product Line Conference, Second Volume, Lero International Science Centre, University of Limerick, Ireland, ISBN 978-1-905952-06-9, 2008, pp. 227-232.
4. Feature Model Constraints Control in Stage Configuration of Software Product Lines- Elham Darmanaki Farahani¹ and Jafar Habibi² ¹Kish International Campus, Sharif University of Technology, Iran ²Department of Computer Engineering, Sharif University of Technology, Iran.
5. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak and A.S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990).
6. Exploring the Commonality in Feature Modeling Notations- Miloslav ŠÍPKA* Slovak University of Technology Faculty of Informatics and Information Technologies Ilkovičova 3, 842 16 Bratislava, Slovakia.
7. Functional Testing of Feature Model Analysis Tools: A Test Suite- Sergio Segura, David Benavides and Antonio Ruiz-Cortés Department of Computer Languages and Systems Av Reina Mercedes S/N, 41012 Seville, Spain {sergiosegura, benavide.
8. Automated Analysis of Feature Models 20 Years Later: A Literature Review- David Benavides, Sergio Segura and Antonio Ruiz-Cortés Dpto. de Lenguajes y Sistemas Informáticos, University of Seville Av. Reina Mercedes s/n, 41012, Seville – Spain

9. D. Benavides, P. Trinidad and A. Ruiz-Cortés, "Automated Reasoning on Feature Models", in: O. Pastor and J. Falcão e Cunha (Eds.): CAiSE 2005, Lecture Notes in Computer Science 3520, Springer-Verlag Berlin Heidelberg, 2005, pp. 491-503.
10. L. Abo, F. Kleinermann, and O. De Troyer. Applying se-mantic web technology to feature modeling. In SAC '09:Proceedings of the 2009 ACM symposium on Applied Com-puting, pages 1252–1256, New York, NY, USA, 2009. ACM.
11. F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors. The description logic hand-book: theory, implementation, and applications. CambridgeUniversity Press, New York, NY, USA, 2003.
12. Alloy analyzer, <http://alloy.mit.edu/>. accessed Jan-uary 2010.
13. https://en.m.wikipedia.org/wiki/Feature_model
14. <https://en.m.wikipedia.org/wiki/Miranda>