

East West University

Logic Based Verification of Software Product Line Feature Model

by

Mirza Faisal Md. Abdul Bari

&

Morium Akter

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

in the

Faculty of Science and Engineering

Department of computer Science and Engineering

December 2014

Declaration

We hereby declare that this submission is our own work and that to the best of our knowledge and belief it contains neither material nor facts previously published or written by another person. Further, it does not contain material or facts which to a substantial extent have been accepted for the award of any degree of a university or any other institution of tertiary education except where an acknowledgement.

(Mirza Faisal Md. A. Bari)

ID: 2010-2-68-015

(MoriumAkter)

ID:2010-2-60-010

Letter of Acceptance

The project entitled “**Logic Based Verification of Software Product Line Feature Model**” submitted by Mirza Faisal Md. Abdul Bari (ID: 2010-2-68-015) and MoriumAkte(ID:2010-2-60-010),to the department of Computer Science and Engineering, East West University, Dhaka, Bangladesh is accepted by the department in partial fulfillment of requirements for the Award of the Degree of Bachelor of Science in Computer Science and Engineering on December 2014

Board of Examiners

Dr. Shamim H. Ripon

Associate Professor & Chairperson

Department of Computer Science and Engineering

East West University, Dhaka-1212, Bangladesh

Acknowledgements

First of all Thanks to ALLAH for the uncountable blessings on us. Thanks to our Supervisor and chairperson Dr. Shamim Hasnat Ripon for providing us this opportunity to test our skills in the best possible manner. He enlightened, encouraged and provided us with ingenuity to transform our vision into reality. East West University, for his encouragement, guidance and counseling. Thanks to our family who helped us out during this project.

Abstract

Feature diagrams are widely used to model product line variant . Formal Verification of variant requirements has gained much interest in the software product line(SPL) community . However, there is a lack of precisely defined formal notation for representing and verifying such models. This report presents an approach to modeling and analyzing SPL variant feature by Logic Based and also First order logic. The logical representation provides a precise and rigorous formal interpretation of the feature diagrams. Logical expressions can be built by modeling variants and their dependencies by using propositional connectives. These expressions can then be validated by any suitable verification tool such as Alloy. A case study of two Feature Model (GPL & Hall Booking System) variant feature model is presented to illustrate the analysis and verification process.

Contents

Acknowledgement

Abstract **i**

List of Figure **ii**

List of Tables

1. Introduction	1-3
1.1 Introduction	1
1.2 Problems and motivation	2
1.3 Objectives	2
1.4 Contribution	3
1.5 Outline	3
2. Background	4-14
2.1 Feature model	4
2.1.1 Feature Model Notations	5
2.1.1.1 Basic Features Models	5
2.1.1.2 Cardinality Based Features Models	6
2.1.1.3 Extended Feature Models	6
2.2 Logical Representation	6
2.2.1 Logical Operators	7
2.3 Propositional Formulas	9
2.4 Domain Engineering And Application Engineering	10
2.5 Alloy Analyzer	11
2.5.1 Short Note About Alloy	12
2.5.2 Syntax and semantics	12
2.5.3 Instances and Meaning	13
3. Case study	15-17
3.1 Introduction of GPL	15
3.1.1 Overview of GPL Feature Tree	15
3.2 Hall Booking System Feature Tree	16

3.2.1 Overview of Hall Booking Feature Tree	17
4. Logical Modeling Of Feature Tree	18-32
4.1 Introduction	18
4.1.1 Logical Expression Of Feature Tree	20
4.2 Set Representation of SPL	20
4.2.1 Feature types	20
4.3 Analysis of feature types	27
5. Model Verification in Alloy	33-39
5.1 Representing Our Feature Model using Alloy	33
5.1.1 Alloy Encoding	34
5.1.2 Semantict Part of the above Mentioned Tree	34
5.2 Alloy Encoding	35
5.2.1 Output Meta model for GPL	35
5.2.3 Output validation for Hall Booking System	36
5.2.5 Output validation for Invalid Conf	37
6. Conclusion	38-39
6.1 Conclusion	38
6.2 Future Work	39
7.Appendix	40-46
A.1 Alloy encoding for GPL	40
A1.1 Meta model for GPL	42
A.2 Alloy encoding for Hall Booking system	42
A2.1 Meta model for Hall Booking system	44
A.3 Invalid Configuration for GPL	44

List Of Table

2.2.1 Truth Table For Negation Operator	7
2.2.2 Truth Table For Conjunction Operator	7
2.2.3 Truth Table For Disjunction Operator	8
2.2.4 Truth Table For Exclusive Operator	8
2.2.5 Truth Table For Implicit Operator	9
2.2.6 Truth Table For Biconditional Operator	9
4.2.1 Truth Table For Mandatory Notation	21
4.2.2 Truth Table For Optional Notation	22
4.2.3 Truth Table For Alternative Notation	22
4.2.4 Truth Table For Optional Alternative Notation	23
4.2.5 Truth Table For Or Notation	24
4.2.6 Truth Table For Optional Or Notation	25
4.2.7 Truth Table For Require Notation	26
4.2.8 Truth Table For Exclude Notation	27

List Of Figure

2.1 Feature Model For Mobile Phone	5
3.1 The Graph Product line Feature Model	15
3.2 Hall Booking System Feature Tree	16
4.1 Notation of The feature Rules	18
4.1.1 Logical Notation of Feature Model	20
4.2.1 Set Representation For Mandatory Feature	21
4.2.2 Set Representation For Optional Feature	21
4.2.3 Set Representation For Alternative Feature	22
4.2.4 Set Representation For Optional Alternative Feature	23
4.2.5 Set Representation For Or Feature	24
4.2.6 Set Representation For Optional Or Feature	25
4.2.7 Set Representation For Require Feature	26
4.2.8 Set Representation For Exclude Feature	27
4.3.1 Require dependency between variants and variation point	28
4.3.2 Exclude dependency between variants and variants point	28
4.3.3 Require dependency between variants and variation point	29
4.3.4 Exclude dependency between variants and variation point	29
4.3.5 Require dependency between variation points	30
4.3.6 Exclude dependency between variation points	31
4.3.7 Exclude dependency between variation points	31
4.3.8 Variation point to variation point and parent-child relation	32
4.3.9 Variant and variation point exclude relation	32
5.1.1 A small part of the GPL	33

5.2.1 For validation or GPL	36
5.2.3 For validation of Hall Booking System	37
5.2.5 Run invalid Configfor GPL feature tree	39
A1.1 Meta model of GPL feature tree	42
A2.1 Meta model of Hall booking system	44

Chapter 1

Introduction

1.1 Introduction

Software product line 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[24]. Software product line engineering can also enable rapid market entry and flexible response, and provide a capability for mass customization.

A software product line 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 [21]. Core assets are the basis for software product line. The core assets often include the architecture, reusable software components, domain models, requirements statements, documentation and specifications, performance model, etc. Different product line members may differ in functional and non-functional requirements, design decisions, run-time architecture and interoperability (component structure, component invocation, synchronization, and data communication), platform, etc. The product line approach integrates two basic processes: the abstraction of the commonalities and variability of the products considered (development for reuse) and the derivation of product variants from these abstractions (development with reuse) [22]. The main idea of software product line is to explicitly identify all the requirements that are common to all members of the family as well as those that varies among products in the family. This implies a huge model that help the stakeholders to be able to trace any design choices and variability decision. A particular product is then derived by selecting the required variants and configuring them according to the product requirements.

Common requirements among all family members are easy to handle and can be integrated into the family architecture and are part of every family member. But problem arises from the variant requirements among family members. Variants are usually modeled using feature diagram, inheritance, templates and other techniques.

In comparison to analysis of a single system, modeling variants adds an extra level of complexity to the domain analysis. Different variants might have dependencies on each other. Tracing multiple occurrences of any variant and understanding their mutual dependencies are major challenges during domain modeling. While each step in modeling variants may be simple but problem arises when the volume of information grows. As a result, the impact of variant becomes ineffective on domain model. Therefore, product customization from the product line model becomes unclear and it undermines the very purpose of domain model.

1.2 Problem and Motivation

Both industry and academia have shown much interest in handling product line in application domains such as business systems, avionics, command and control systems etc. Today most of the effort in product line development are relating to architecture [23], detail design and code. Common requirements among all family members are easy to handle as they simply can be integrated into the family architecture and are part of every family member. But problem arises from the variant requirements among family members.

In a product line, currently variants are modeled using feature diagram, inheritance, templates and other techniques. In comparison to analysis of a single system, modeling variants adds an extra level of complexity to the domain analysis. In any product line model, the same variant has occurrences in different domain model views. Different variants have dependencies on each other. Tracing multiple occurrences in different model views of any variant and understanding the mutual dependencies among variants are major challenges during domain modeling. While each step in modeling variant may be simple but problem arises when the volume of information grows. When the volume of information grows the domain models become difficult to understand. The main problems are the possible explosion of variant combinations, complex dependencies among variants and difficulty in tracing variants from the domain model down to the specification of a particular product. As a result, the impact of variant becomes ineffective on domain model. Therefore, product customization from the product line model becomes unclear and it undermines the very purpose of domain model.

1.3 Objectives

In developing product line, the variants are to be managed in domain engineering phase, which scopes the product line and develops the means to rapidly produce the members of the family. It serves two distinct but related purposes, firstly it can record decisions about the product as a whole including identifying the variants for

each member and secondly ,it can support application engineering by providing proper information and mechanism for the required variants during product generation

- The objective of this work is to provide an approach for modeling variants in the domain model of a product line .This model carries all the variant related information like specifications ,origin of variants and interdependencies etc.
- Defining the search table involves manual handling of variants ,formal verification is not directly admissible for such approach. Our objective is to logically representation of feature model facilitating the development of decision table in formally sound way.
- Our plan is perform these verification by using our logical representation.

1.4 Contribution

In order to conduct out experiment we use a case study of Graph product line(GPL) and hall booking system by analyzing and modeling the variants as well as the variants dependencies.

- We define six types of logical notation to represent all the parts in a feature model. Set representation logic has been for this purpose. This notations can be used to define all possible scenarios of a feature model.
- Analyzing the feature model considering the various scenarios the feature model and we define a set of rules which can be used to verify the feature model .
- We use Alloy tools for checking the valid or invalid feature model. Alloy use first order logic and encode our logical definitions into Alloy and validity of the logical verification

1.5 Outline

The report is organized as follows-

In chapter 2 we gave a brief overview of the feature model, feature model notations and logical representation of the feature model and describe some logical operators, domain activities and give a brief review of the model analyzer named alloy with an example model.

In Chapter 3 We have gave an overview of GPL and an hall booking feature tree.

In chapter 4 we discuss about the logical representation and describe their logical relations and analyze the set representation

Chapter 5 We presents the alloy representation of the logical notations. We illustrate the steps how the logical representations are encoded into alloy and how the verification has been preformed. Chapter 6Concludes the thesis by summarizing our work . Finally we outline our future plan

Chapter 2

Background

2.1 Feature Model

Feature modeling is a key approach to capturing and managing the common and variable features of systems in a system family or a product line. In the early stages of software family development, feature models provide the basis for scoping the system family by recording and assessing information such as which features are important to enter a new market or remain in an existing market, which features incur a technological risk, what is the projected development cost of each feature, and so forth [1]. Later, feature models play a central role in the development of a system family architecture, which has to realize the variation points specified in the feature models [2][3]. In application engineering, feature models can drive requirements elicitation and analysis. Knowing which features are available in the software family may help customers decide which features their system should support. Knowing which desired features are provided by the system family and which have to be custom-developed helps to better estimate the time and cost needed for developing the system.

A software pricing model could also be based on the additional information recorded in a feature model. Feature models also play a key role in generative software development [2][4]. Generative software development aims at automating application engineering based on system families: a system is generated from a specification written in one or more textual or graphical domain-specific languages (DSLs). In this context, feature models are used to scope and develop DSLs [2][5], which may range from simple parameter lists or feature hierarchies to more sophisticated DSLs with graph-like structures. Feature modeling was proposed as part of the Feature-Oriented Domain Analysis (FODA) method [6], and since then, it has been applied in a number of domains including telecom systems [10][11], template libraries [2], network protocols [8], and embedded systems [9]. Based on this growing experience, a number of extensions and variants of the original FODA notation have been proposed [6][7][9][10].

Example

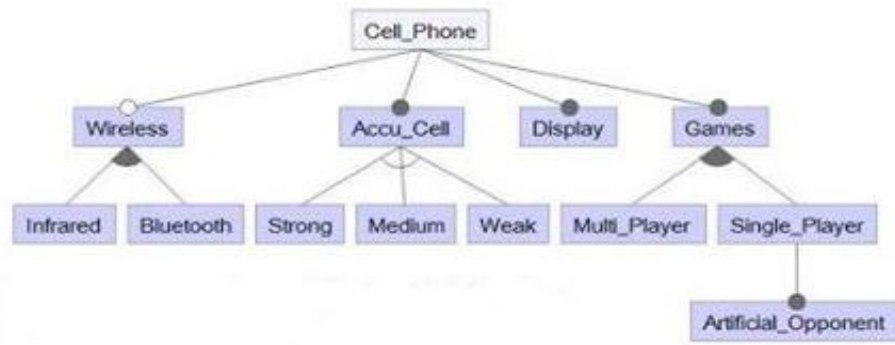


Fig2.1: Feature model for mobile phone

2.1.1 Feature Modeling Notations

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

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

2.1.1.1 Basic Feature Models

Czarnecki's notation proposes four relations, namely: mandatory, optional, alternative and or-relation.[25] In these relations, there is always a parent feature and one (in the case of mandatory and optional relations) or more (in the case of alternative and or-relation) child features.

- Mandatory - child feature is required.
- Optional – child feature is optional.
- Or – at least one of the sub-features must be selected.
- Alternative (xor) – one of the sub-features must be selected

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.

2.1.1.2 Cardinality-Based Feature Models

Some authors propose extending basic feature models with UML-like multiplicities of the form $[n,m]$ with n being the lower bound and m the upper bound. These are used to limit the number of sub-features that can be part of a product whenever the parent is selected.[26]If the upper bound is m the feature can be cloned as many times as we want (as long as the other constraints are respected). This notation is useful for products extensible with an arbitrary number of components.

2.1.1.3 Extended Feature Models

Current proposals only deal with characteristics related to the functionality offered by an SPL (functional features). Thus, there exists no solid proposal for dealing with the remaining characteristics, also called extra-functional features. There are several concepts that we would like to clarify before analyzing current proposals and framing our contribution:

- Feature: a prominent characteristic of a product. Depending on the stage of development, it may refer to a requirement (if products are requirement documents), a component in an architecture [26] (if products are component architectures) or even to pieces of code (if products are binary code in a feature oriented programming approach) of an SPL.
- Attribute: the attribute of a feature is any characteristic of a feature that can be measured. *Availability* and *cost* are examples of attributes of the *Service* feature. *Latency* and *bandwidth* may be examples of attributes of an Internet connection.
- Attribute domain: the space of possible values where the attribute takes its values. Every attribute belongs to a domain. It is possible to have discrete domains (e.g: Integers,Booleans , enumerated) or continuous domains (e.g.: real).
- Extra-functional feature: a relation between one or more attributes of a feature. For instance: *bandwidth* Latency Availability and so on. These relations are associated to a feature.

2.2 Logical Representation

Logic has been studied since the classical Greek period (600-300BC). The Greeks, most notably Thales, were the first to formally analyze the reasoning process. Aristotle (384-322BC), “the father of logic”, and many other Greeks searched for universal truths that were irrefutable. A second great period for logic came with the

use of symbols to simplify complicated logical arguments. Gottfried Leibniz (1646-1716) began this work at age 14, but failed to provide a workable foundation for symbolic logic. George Boole (1815-1864) is considered the “father of symbolic logic”. He developed logic as an abstract mathematical system consisting of defined terms (propositions), operations (conjunction, disjunction, and negation), and rules for using the operations. Boole’s basic idea was that if simple propositions could be represented by precise symbols, the relation between the propositions could be read as precisely as an algebraic equation. Boole developed an “algebra of logic” in which certain types of reasoning were reduced to manipulations of symbols.

2.2.1 Logical operators

1. Negation Operator: “not”, has symbol “ \neg ” :

Example: p: This book is interesting. Then $\neg p$ can be read as “This book is not interesting”.

2.2.1 Truth Table for Negation operator

P	$\neg P$
T	F
F	T

The negation operator is a unary operator which, when applied to a proposition p, changes the truth value of p. That is, the negation of a proposition p, denoted by $\neg p$, is the proposition that is false when p is true and true when p is false.

2. Conjunction Operator: “and”, has symbol “ \wedge ”.

Example:

p: This book is interesting.

q: I am staying at home. $p \wedge q$: This book is interesting and I am staying at home.

2.2.2 Truth Table for conjunction operator

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

The conjunction operator is the binary operator which, when applied to two propositions p and q, yields the proposition “p and q”, denoted $p \wedge q$. The conjunction $p \wedge q$ of p and q is the proposition that is true when both p and q are true and false otherwise.

3. Disjunction Operator: inclusive “or”, has symbol “ \vee ”.

Example:

p: This book is interesting
 q: I am staying at home.
 p_∨q: This book is interesting, or I am staying at home.

2.2.3 Truth Table for Disjunction operator

P	Q	P ∨ Q
T	T	T
T	F	T
F	T	T
F	F	F

The disjunction operator is the binary operator which, when applied to two propositions p and q, yields the proposition “p or q”, denoted p_∨q. The disjunction p_∨q of p and q is the proposition that is true when either p is true, q is true, or both are true, and is false otherwise

4. Exclusive Or Operator: “xor”, has symbol “⊕”.

Example:

p: This book is interesting
 q: I am staying at home.
 p_⊕q: Either this book is interesting or I am staying at home, but not both.

2.2.4 Truth table for Exclusive operator

P	Q	P ⊕ Q
T	T	F
T	F	T
F	T	T
F	F	F

The exclusive or is the binary operator which, when applied to two propositions p and q yields the proposition “p xor q”, denoted p_⊕q, which is true if exactly one of p or q is true, but not both. It is false if both are true or if both are false.

5. Implication Operator :“if...then...”, has symbol “⇒” .

Example:

p: This book is interesting.
 q: I am staying at home.
 p⇒q: If this book is interesting, then I am staying at home.

Truth Table:

2.2.5 Truth table for Implicit Operator

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

The implication $p \Rightarrow q$ is the proposition that is often read as “if p then q”. If “p then q” is false precisely when p is true but q is false.

6. Biconditional Operator: “if and only if”, has symbol “ \Leftrightarrow ”

Example:

p: This book is interesting.

q: I am staying at home.

p,q: This book is interesting if and only if I am staying at home.

2.2.6 Truth table for Biconditional Operator

P	Q	$P \Leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

The bi-conditional statement is equivalent to $(p \Rightarrow q) \wedge (q \Rightarrow p)$.

In other words: For p,q to be true we must have both p and q true.

2.3 Propositional Formulas

Mannion was the first to connect propositional formulas to product-lines [14]; we show how his results integrate with those. A *propositional formula* is a set of Boolean variables and a propositional logic predicate that constrains the values of these variables. Besides the standard \wedge , \vee , \neg , \Rightarrow , and \Leftrightarrow operations of propositional logic, we also use $\text{choose}_1(e_1 \dots e_k)$ to mean at most one of the expressions $e_1 \dots e_k$ is true. More generally, $\text{choose}_{n,m}(e_1 \dots e_k)$ means at least n and at most m of the expressions $e_1 \dots e_k$ are true, where $0 \leq n \leq m \leq k$. A grammar is a compact representation of a propositional formula. A variable of the formula is either: a token, the name of a non-terminal, or the name of a pattern. For example, the production:

$r : A B :: P1$
 $| C [r1] :: P2 ;$

has seven variables: three $\{A, B, C\}$ are tokens, two are non-terminals $\{r, r1\}$, and two are names of patterns $\{P1, P2\}$. Given these variables, the rules for mapping a grammar to a propositional formula are straightforward.

2.4 Domain Engineering and Application Engineering

Domain is an area of knowledge that uses common concepts for describing phenomena, requirements, problems, capabilities, and solutions that are of interest to some stakeholders. A domain is usually associated with well-defined or partially defined terminology. This terminology refers to the basic concepts in that domain, their definitions (i.e., their semantic meanings), and their relationships. It sometime also refers to behaviors that are desired, forbidden, or perceived within the domain. Domain engineering is a set of activities that aim at developing, maintaining, and managing the creation and evolution of domains. Domain engineering has become of special interest to the information systems and software engineering communities for several reasons. These reasons include, in particular, the need to manage increasing requirements for variability of information and software systems (reflecting variability in customer requirements); the need to minimize accidental complexity when modeling the variability of a domain; and the need to obtain, formalize, and share expertise in different, evolving domains.

Domain engineering as a discipline has practical significance as it can provide methods and techniques that may help reduce time-to-market, product cost, and project risks on one hand, and help improve product quality and performance on a consistent basis on the other hand. It is used, researched, and studied in various fields, the main ones of which are Software Product Line Engineering, Domain-Specific Language Engineering, and Conceptual Modeling & Knowledge Engineering. Domain engineering is designed to improve the quality of developed software products through reuse of software artifacts. Domain engineering shows that most developed software systems are not new systems but rather variants of other systems within the same field. As a result, through the use of domain engineering, businesses can maximize profits and reduce time-to-market by using the concepts and implementations from prior software systems and applying them to the target system. The reduction in cost is evident even during the implementation phase. One study showed that the use of domain-specific languages allowed code size, in both number of methods and number of symbols, to be reduced by over 50%, and the total number of lines of code to be reduced by nearly 75%.

Domain engineering focuses on capturing knowledge gathered during the software engineering process. By developing reusable artifacts, components can be reused in new software systems at low cost and high quality. Because this applies to all phases of the software development cycle, domain engineering also focuses on the three primary phases: analysis, design, and implementation, paralleling application engineering. This produces not only a set of software implementation components relevant to the domain, but also reusable and configurable requirements and designs.

An application engineer plans the design and implementation of technology products like specialty industry equipment or computer programs. He or she works together with a company's manufacturing, sales, and customer service departments. Companies typically require this type of worker to have a four-year degree along with years of field experience. He or she should have good communication, math and teamwork skills.

Domain engineering, like application engineering, consists of three primary phases: analysis, design, and implementation. However, where software engineering focuses on a *single* system, domain engineering focuses on a family of systems. A good domain model serves as a reference to resolve ambiguities later in the process, a repository of knowledge about the domain characteristics and definition, and a specification to developers of products which are part of the domain.

2.5 Alloy Analyzer

Alloy is a language for describing structures and a tool for exploring them. It has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks.

An Alloy model is a collection of constraints that describes (implicitly) a set of structures, for example: all the possible security configurations of a web application, or all the possible topologies of a switching network. Alloy's tool, the Alloy Analyzer, is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples. Structures are displayed graphically, and their appearance can be customized for the domain at hand.

At its core, the Alloy language is a simple but expressive logic based on the notion of relations, and was inspired by the Z specification language and Tarski's relational calculus. Alloy's syntax is designed to make it easy to build models incrementally, and was influenced by modeling languages (such as the object models of OMT and UML). Novel features of Alloy include a rich subtype facility for factoring out common features and a uniform and powerful syntax for navigation expressions. Alloy and Alloy Analyzer were developed by Daniel Jackson's group at MIT.[17]

Two Statements About Alloy

“The examples and exercises, if given time, thought, and effort, can make better designers of all of us, as Alloy is a powerful force-multiplier in the war on bugs... Jackson's Software Abstractions has my highest recommendation. It is being put to immediate use in my group's venue of software-based safety-critical systems”—George Hacken, Computing Reviews [17].

“Systems like Alloy should be in the toolbox of all software designers and developers, so such a comprehensive book on this topic is very welcome.”
—Anthony M. Sloane, Journal of Functional Programming[17]

2.5.1 Short Note About Alloy

- Alloy is an object oriented modeling language
- Alloy has formal syntax and semantics
- Alloy specifications can be written in ASCII
- Alloy also has a visual language similar to UML class diagrams
- Alloy has a constraint analyzer which can be used to automatically analyze properties of Alloy models

2.5.2 Syntax and Semantics

Sig: Signatures that are used for defining new types and constraints. Each signature denotes a set of objects, which are associated to other objects by relation declared in the signature. A signature paragraph introduces a type and a collection of relations such as- person, feature, person ,Relation and etc. In this alloy sets- sig declarations defining the signature. The sig notations –

```
sig Relation{
parent : Name
child: set Name
Type:Type}
```

A FM include various types of relations . A sig can represent the several feature at a time. So-

```
abstract sig Type {}
One sig v,v1,v11,v2,v21 extends Type {}
```

Predicate are used to package reusable formulas. The subsequent fragment declares the relation . Pred declarations defining the predicate. The pred notations-

```
pred own Grandpa[p:person ] {
p in grandpas[p]
}
```

A configuration relation ,which contains a set of feature names (selected for a given software product) ,is represented by the following sig. It is important to notice that

this signature is a datatype. A data type has an implicit generates all possible combination, as stated by the fact configdatatype.

```
sigconf{
  Value: set Name
}
factconfigdatatype{
  all n:set Name | some c:conf | c.value=n
}
```

Multiplicity is used to define the number of objects required. It has some kinds like

set(zero or more)

one (exactly one)

lone(zero or none)

some (one or more)

Those examples:-

```
one sig A{} // A is singleton set
lone sig B{} // B is a singleton or empty
some sig C{} // C is a non-empty set
```

2.5.3 Instances and Meaning

A model's meaning is several collections of instances. An instance is a binding of values to variables. Typically, a single instance represents a state, or a pair of states (corresponding to execution of an operation), or an execution trace. The language has no built-in notion of state machines, however, so an instance need not be represented of these things.

The collections of instances assigned to a model are:

- A set of signatures and their fields, and they bind values to them that make this of core instances associated with the facts of the model, and the constraints implicit in the signature declarations. These instances have as their variables the signature and their fields.
- For each function or predicate, a set of those instances for which the facts and declaration constraints of the model as a whole are true, and additionally the constraint of the function or predicate are true. The variables of these instances are those of the core instances, extended with the arguments of the function or predicate.

- For each assertion, a set of those instances for which the facts and declaration constraints of the model as a whole are true, but for which the constraint of the assertion is false.
- A model without any core instances is inconsistent, and almost certainly erroneous. A function or predicate without instances is likewise inconsistent, and is unlikely to be useful. An assertion is expected not to have any instances: the instances are counterexamples, which indicate that the assertion does not follow from the facts.
- The Alloy Analyzer finds instances of a model automatically by search within finite bounds (specified by the user as a scope). Because the search is bounded, failure to find an instance does not necessarily mean that one does not exist. But instances that are found are guaranteed to be valid.

Chapter 3

Case Study

3.1 Introduction of GPL

We offer the following domain as an example to compare and contrast approaches to the definition and implementation of product-line architectures. We believe this a good example, because it deals with a classical domain whose algorithms (i.e., graph algorithms) are common-knowledge to computer scientists. This relieves readers of burden and overhead that accompanies the understanding of an unfamiliar domain.

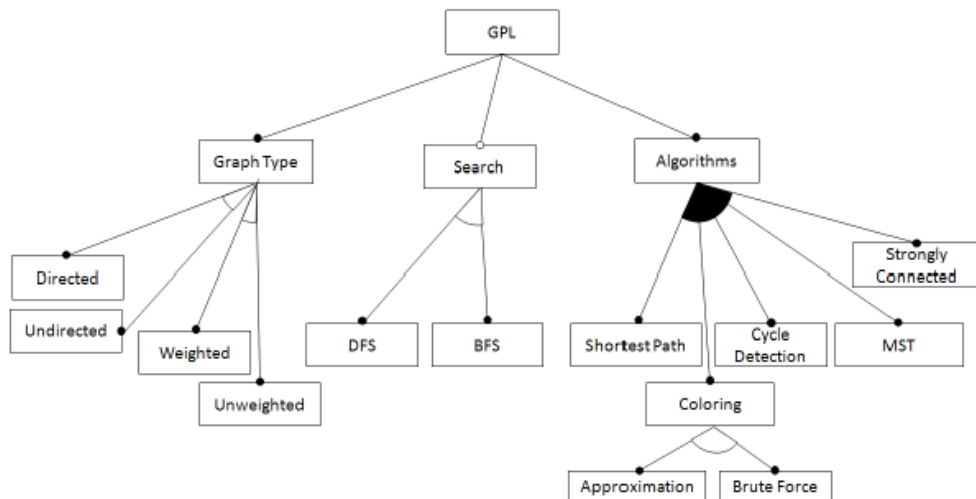


Figure:3.1The graph product line feature model

3.1.1 Overview of GPL Feature Tree

A possible feature diagram for the graph library is shown in the root is labeled with *GraphLibrary* to represent a graph product (That is , a graph library). It has Two mandatory child feature *GraphType*, *Algorithm* because each graph library has to implement an graph type, which is either *Directed* or *Undirected* and *Weighted* or *Unweighted*. Furthermore, one other child features of the root is optional *Search*. *Search* strategies may be either *breadth-first search*(*BFS*) or *depth -first search*

(DFS). Since it is optional, either zero or one Feature may be present in a graph product. *Algorithm* offers a selection of graph algorithms as child features & the child Features are *ShortestPath*, *Coloring*, *Cycle Detection*, *MST* and *StronglyConnected*. Since they are in 'Or' feature, either one or more feature may be present in graph product line. In our example the *algorithm* for *coloring* has two alternative implementations, *BruteForce* and *approximation*. Some non-local conditions are modeled as explicit Boolean constraints – for example, *minimal spanning trees (MST)* makes only sense for *weighted* graphs, and *ShortestPath* can be computed for directed graphs only.

3.2 Hall Booking System Feature Tree

Hall Booking software is online/manual booking software for room and conference facility reservations. This software makes booking more efficient for clients, staff, and conference facilities. It simplify the process, maximize capacity, and provide a seamless service from first click to confirmation

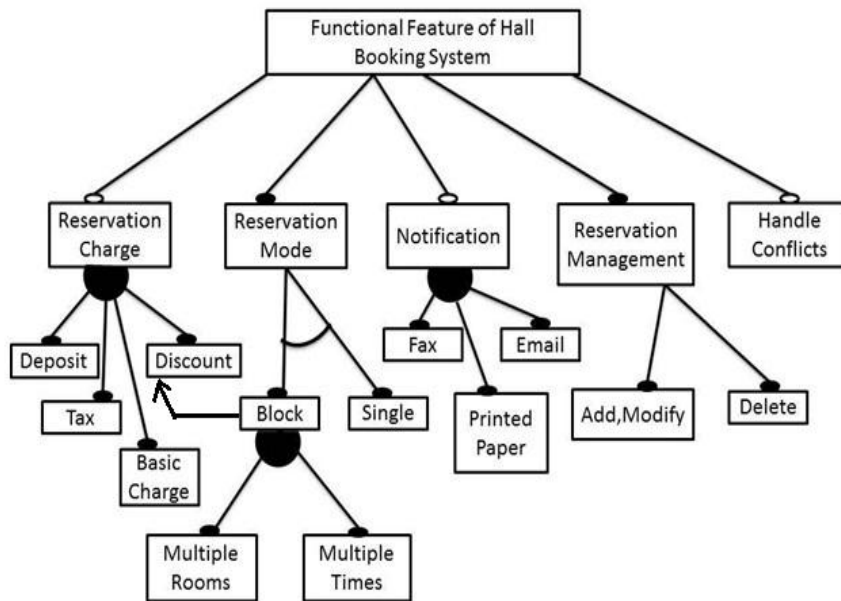


Fig:3.2 Hall Booking System Feature Tree

3.2.1 Overview of Hall Booking System

We use Hall Booking System family to illustrate our variability modeling mechanism. The system is used in academic institutions to reserve tutorial rooms and lecture halls, at companies to reserve meeting rooms, and at hotels to reserve rooms and conference facilities, etc. In another sense, the system can be used for either academic or non-academic purposes. Users can manage their own reservation with the system. The main purpose and the core functionality are similar across the Hall Booking System family; however, there are many variants on the basic theme. One of the basic variants is the charging of the booking system. Whenever the system is used for academic purposes, no charge is needed for booking halls, whereas there may be a need to charge for booking halls in other areas. In some systems, there are facilities available for seasonal booking as well as multiple bookings.

The Root of this system is *Functional Feature of Hall Booking System*. It has Five Direct Feature. Two of them are Mandatory Feature and they are *Reservation Mode* and *Reservation Management*. Three of the features are Optional and they are *Reservation Charge*, *notification* and *Handle Conflict*. Under *Reservation Charge* feature there has four child feature and they are *Deposit*, *Tax*, *Basic Charge* and *Discount* and all of them are in Or relationship with their parent feature *Reservation Charge*. The *Block* and *Single* feature are in Alternative Relationship with their Parent Feature *Reservation Mode* and Under *Block* Parent feature *Multiple Rooms* and *Multiple Time* are in Or relationship. On the Other hand *Block* under *Reservation Mode* and *Discount* feature under *Reservation Charge* are in Require Relationship. Notification contain three Child feature in Or relation and they are *Print Paper*, *Email* and *Fax*. *Add Modify* and *Delete* are two mandatory Feature under the *Reservation Management* Parent Feature.

Chapter 4

Logical Modeling of Feature Tree

4.1 Introduction

Logic has been studied since the classical Greek period(600-300BC). The Greeks , most notableThalas, were the first to formally analyze the reasoning process. Aristotle (384-322BC) , “The father of logic”, and many other Greeks searched for universal truths that were irrefutable. A second great period for logic came with the use of symbols to simplify complicated logical arguments. Gottfried Leibniz (1646-1716) began this work at age 14, but failed to provide a workable foundation for symbolic logic. George Boole (1815-1864) is considered the “Father of symbolic logic” He developed logic as an abstract mathematical system consisting of defined terms(Proposition), operations(Conjunction, Disjunction and negation), and rules for using the operation. Boole’s basic idea was that if simple propositions could be represented by precise symbols, the relation between the propositions could be read as precisely as an algebraic equation. Boole developed an “Algebra of logic” in which certain types of reasoning were reduced to manipulations of symbols.

A feature model is a hierarchically arranged set of features. The relationships between a parent (or variation point) feature and its child features (Variations) are categorized as follows:

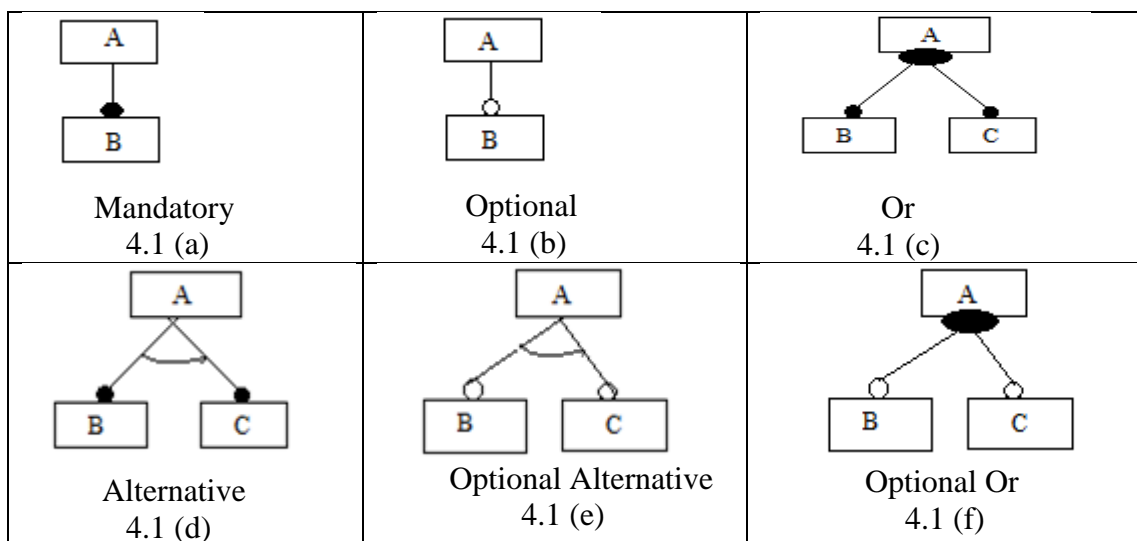


Fig: 4.1 Notations of the feature rules

- **Mandatory :** A mandatory feature is included if its parent feature is included. Mandatory feature is represented by a small circle on the child node. A filled bullet denotes a mandatory (In fig : 4.1 (a)) feature and features that are required.
- **Optional:** An optional Feature may or may not be include if its parent is included.Optional Feature is represented by a small circle on the child node. A empty bullet denotes(In fig : 4.1 (b)) a optional feature and features that are optional .The set notation for optional feature is...
- **OR Feature:** At least one from a set of or feature is included when parent is included and one or more features can be selected when the parent feature appears. Feature is represented by a filled triangle (In fig : 4.1 (c)) denotes the or Feature.
- **Alternative:** One and only one feature from a set of alternative features are included when parent feature is included that means exactly one sub-feature must be selected. Feature is represented by a unfilled(In fig : 4.1 (d)) triangle denotes the alternative.
- **Optional alternative:** One feature from a set of alternative features may or may not be included if parent in included. Feature is represented by a unfilled triangle and empty bullets(In fig : 4.1 (e)) denotes the optional alternative.
- **Optional or:** One or more optional feature may be included if the parent is included. Optional Or Feature is represented by a filled triangle (In fig : 4.1 (f)) and filled bullets denotes the optional or.

4.1.1 Logical expression of feature tree

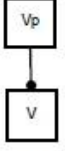
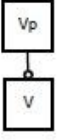
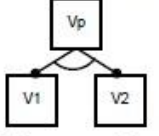
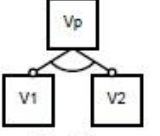
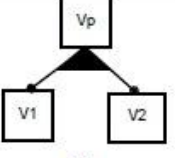
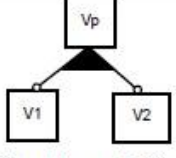
Type	Logic Expression	Type	Logic Expression
 Mandatory	$v_p \Leftrightarrow v$	 Optional	$v \Rightarrow v_p$
 Alternative	$v_p \Leftrightarrow (v_1 \oplus v_2)$	 Optional Alternative	$(v_1 \oplus v_2) \Rightarrow v_p$
 Or	$v_p \Leftrightarrow (v_1 \vee v_2)$	 Optional Or	$(v_1 \vee v_2) \Rightarrow v_p$

Fig 4.1.1: Logical notations of feature model

4.2 Set representation of SPL(Software Product Line)

4.2.1 Feature Types

If we want to represent SPL(Software Product Line) feature rules by set then we have to think the whole element situated in a set “conf”.

Mandatory: We assume that “x” and “A” are two elements inconf Set. Now for define the mandatory feature For all “x” and “A” where “x” and “A” belongs to Set Conf. Then both are Dependent to each other. That means where “x” is present “A” must be present and also where “A” is present “x” must be present. One cant be imagined except another.

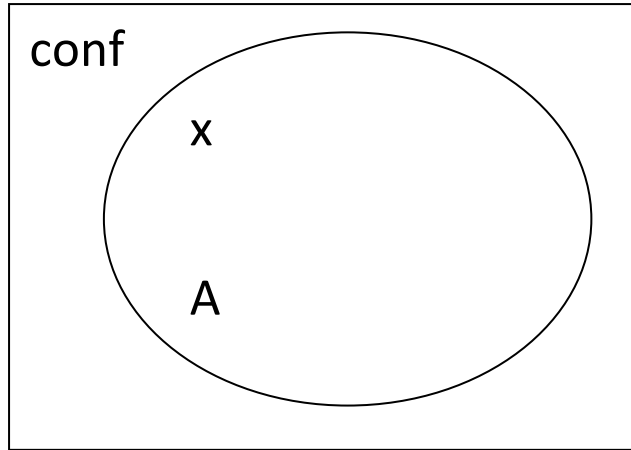


Fig:4.2.1 Set Representation for Mandatory Feature

$$\forall x,A. x \in \text{conf} \wedge A \in \text{conf} \wedge x \Leftrightarrow A$$

4.2.1 Truth table for Mandatory Feature

X	A	$X \Leftrightarrow A$
T	T	T
T	F	F
F	T	F
F	F	F

Optional: We assume that “x” and “A” are two elements in confSet. “x” and “A” are two elements in this Set. Now for define the optional feature For all “x” and “A” where “x” and “A” belongs to Set Conf. Then “x” is Dependent to “A”. That means where “x” is present “A” must be present but where “A” is present “x” may or may not be present.

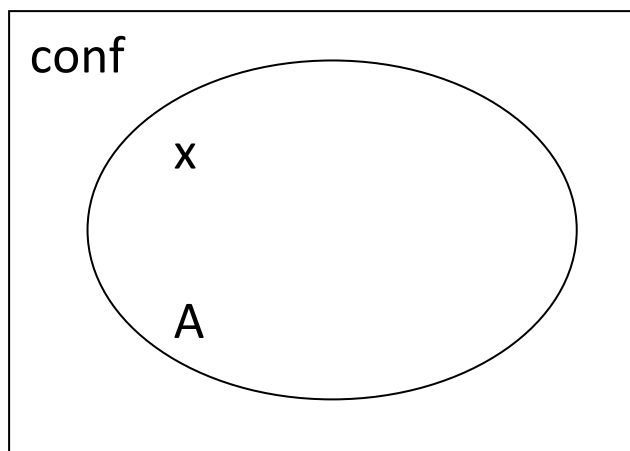


Fig:4.2.2 Set Representation for Optional Feature

$$\forall x,A. x \in \text{conf} \wedge A \in \text{conf} \wedge x \Leftrightarrow A$$

4.2.2 Truth Table For Optional Feature

X	A	$X \Rightarrow A$
T	T	T
T	F	F
F	T	T
F	F	T

Alternative: We assume that “x” ,”y”and “A” are three elements inconf Set. Now for define the Alternative feature For all “x” ,”y” and “A” where “x” ,”y” and “A” belongs to Set Conf. Then if “A”is included to feature then “x” or “y”must be included in feature . .sign represent the XOR \oplus (any one must be selected) relationship.”A” cantbe imagined except any of them.

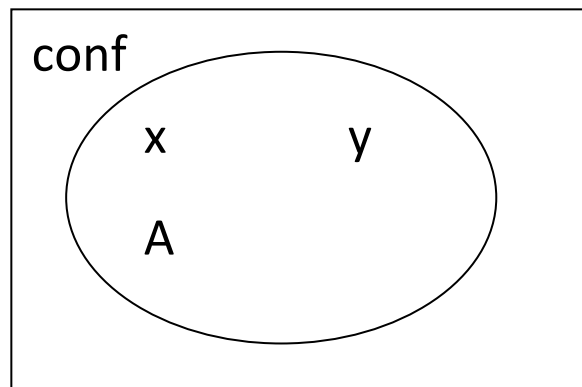


Fig:4.2.3 Set Representation for Alternative Feature

$$\forall x,y,A. x \in \text{conf} \wedge y \in \text{conf} \wedge A \in \text{conf} \wedge x \oplus y \Leftrightarrow A$$

4.2.3 Truth Table For Alternative Feature

X	Y	A	$X \oplus Y$	$(X \oplus Y) \Leftrightarrow A$
T	F	T	T	T
T	F	F	T	F
T	T	T	F	F
T	T	F	F	T
F	F	T	F	F
F	F	F	F	T
F	T	T	T	T
F	T	F	T	F

Optional Alternative: We assume that “x” ,”y” and “A” are three elements in conf Set. Now for define the Alternative feature For all “x” ,”y” and “A” where “x” ,”y” and “A” belongs to Set Conf. Then if “A” is included to feature then “x” or “y” may or may not be included in feature . sign represent the XOR \oplus (any one must be selected) relationship.

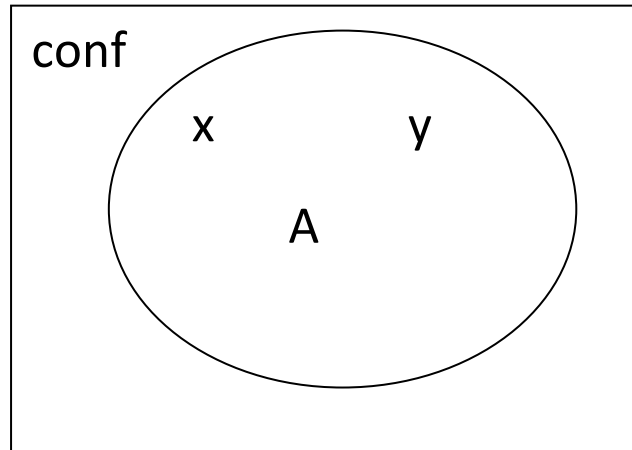


Fig:4.2.4 Set Representation for optional Alternative Feature

$$\forall x,y,A. x \in \text{conf} \wedge y \in \text{conf} \wedge A \in \text{conf} \wedge x \oplus y \Leftrightarrow A$$

4.2.4 Truth Table For Optional Alternative Feature

X	Y	A	$X \oplus Y$	$(X \oplus Y) \Rightarrow A$
T	F	T	T	T
T	F	F	T	F
T	T	T	F	T
T	T	F	F	T
F	F	T	F	T
F	F	F	F	T
F	T	T	T	T
F	T	F	T	F

Or Feature: 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. Now for set representation, We assume that “x” ,”y” and “A” are three elements in conf Set. Now for define the Or feature For all “x” ,”y” and “A” where “x” ,”y” and “A” belongs to Set Conf. Then if “A” is included to feature then “x” or “y” or may both included in feature .But one child feature must be included if the parent feature included. “ \vee ” sign represent the Or (any one must be selected) relationship.

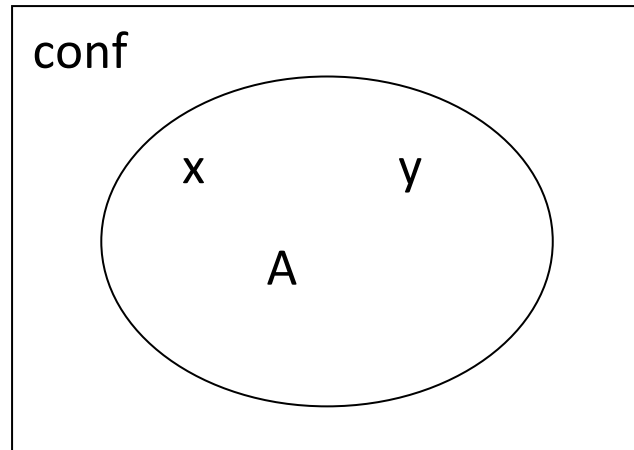


Fig:4.2.5 Set Representation for Or Feature

$$\forall x,y,A. x \in \text{conf} \wedge y \in \text{conf} \wedge A \in \text{conf} \wedge (x \vee y) \Leftrightarrow A$$

4.2.5 Truth Table For Or Feature:

X	Y	A	$X \vee Y$	$(X \vee Y) \Leftrightarrow A$
T	F	T	T	T
T	F	F	T	F
T	T	T	F	F
T	T	F	F	T
F	F	T	F	F
F	F	F	F	T
F	T	T	T	T
F	T	F	T	F

Optional 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. Now for set representation, We assume that “x” ,”y” and “A” are three elements in Conf Set. Now for define the Or feature For all “x” ,”y” and “A” where “x” ,”y” and “A” belongs to Set Conf. Then if “A” is included to feature then “x” or “y” or may both included in feature .But it is not compelled to include any child feature must be included if the parent feature included. “v” sign represent the Or (any one must be selected) relationship.

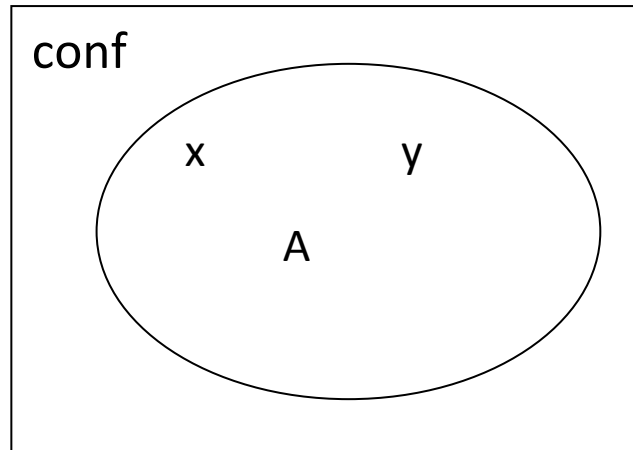


Fig:4.2.6 Set Representation for Optional Or Feature

$$\forall x,y,A. x \in \text{conf} \wedge y \in \text{conf} \wedge A \in \text{conf} \wedge (x \vee y) \Leftrightarrow A$$

4.2.6 Truth Table for Optional Or feature

X	Y	A	$X \vee Y$	$(X \vee Y) \Rightarrow A$
T	F	T	T	T
T	F	F	T	F
T	T	T	F	T
T	T	F	F	T
F	F	T	F	T
F	F	F	F	T
F	T	T	T	T
F	T	F	T	F

Require: Two child feature is in require relationship if one child feature is dependent to another child feature. But the both child feature must belongs to different Parent feature. Now if we want to represent the require relationship set then, we assume that “x”, “y”, “conf1” and “conf2” are the elements of “conf” set but here “x” is belongs to “conf1” set and “y” is belongs to “conf2” set and both of them (“conf1” and “conf2”) are belongs to set “conf” set. It should mention that we are assuming “conf1” and “conf2” set represent the parent(Variant) feature of “x” and “y” variation point(child feature) and also “x”, “y” both cant be in same variation point. So now “x” is strongly dependent in “y”. That means if “x” child feature is included “y” must be included. “x” cant be imagined without “y”.

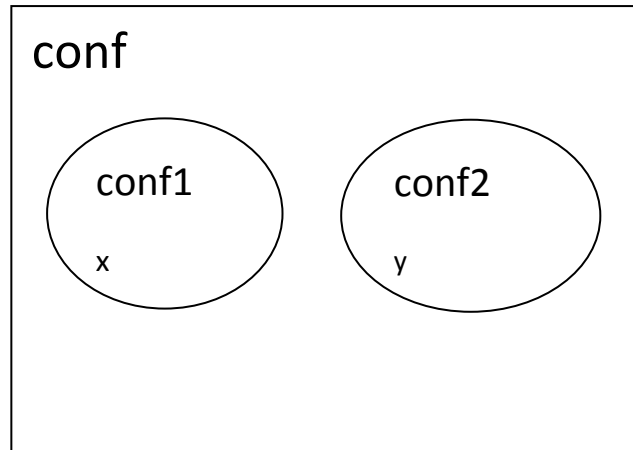


Fig:4.2.7 Set Representation for Require Feature

$\forall x,y,conf1,conf2. x \in conf1 \wedge y \in conf2 \wedge conf1 \subseteq conf \wedge conf2 \subseteq conf \wedge x \notin conf2 \wedge y \notin conf1 \wedge (x \Rightarrow y)$

4.2.7 Truth Table for Require feature

X	Y	A	$X \vee Y$	$(X \vee Y) \Rightarrow A$
T	F	T	T	T
T	F	F	T	F
T	T	T	F	T
T	T	F	F	T
F	F	T	F	T
F	F	F	F	T
F	T	T	T	T
F	T	F	T	F

Exclude: Two child feature is in Exclude relationship if one child feature is conflicted to another child feature. But the both child feature must belongs to different Parent feature. Now if we want to represent the require relationship set then, we assume that “x,”y,”conf1” and “conf2” are the elements of “conf” set but here “x” is belongs to “conf1” set and “y” is belongs to “conf2” set and both of them (“conf1” and “conf2”) are belongs to set “conf” set. It should mention that we are assuming “conf1” and “conf2” set represent the parent(Variant) feature of “x” and “y” variation point(child feature) and also “x”, “y” both cant be in same variation point. So now “x” is strongly conflicted with “y”. That means if “x” child feature is included “y” can’t be included.

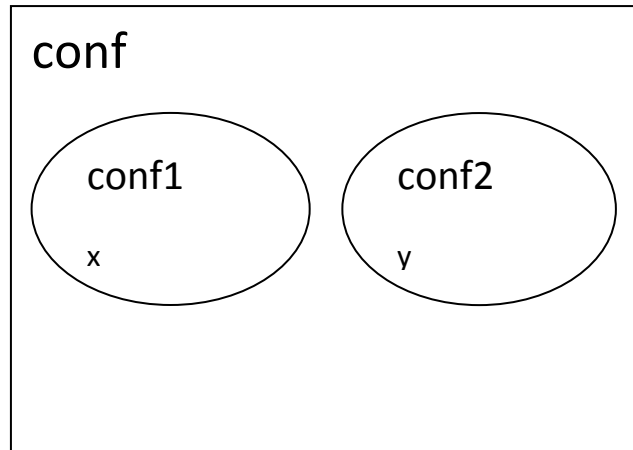


Fig:4.2.8 Set Representation for Exclude Feature

$\forall x,y,conf1,conf2. x \in conf1 \wedge y \in conf2 \wedge conf1 \subseteq conf \wedge conf2 \subseteq conf \wedge x \notin conf2 \wedge y \notin conf1 \wedge (x \oplus y)$

4.2.8 Truth Table for Exclude feature

X	Y	A	$X \oplus Y$	$(X \oplus Y) \Rightarrow A$
T	F	T	T	T
T	F	F	T	F
T	T	T	F	T
T	T	F	F	T
F	F	T	F	T
F	F	F	F	T
F	T	T	T	T
F	T	F	T	F

4.3 Analysis Of Feature Type

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.

4.3.1 Scenario 1

In Fig 4.3.1, v1 and v2 are variants (and variation points) and there is a require dependency between them. Here v2 is selected whenever v1 is selected.

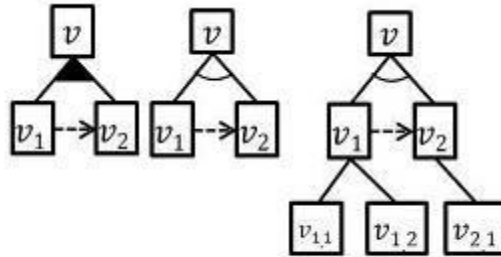


Fig 4.3.1: Require dependency between variants and between variation points

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

$$\forall v_1, v_2 \cdot \text{type}(v_1, \text{variant}) \wedge \text{type}(v_2, \text{variant}) \wedge \text{require } v _v(v_1, v_2) \wedge \text{select}(v_1) \Rightarrow \text{select}(v_2)$$

$$v_1, v_2 \cdot \text{type}(v_1, \text{variation point}) \wedge \text{type}(v_2, \text{variation point}) \wedge \text{requirevp_vp}(v_1, v_2) \wedge \text{select}(v_1) \Rightarrow \text{select}(v_2)$$

where $\text{type}(v_i, \dots)$ indicates whether v_i is a variant or variation point, $\text{select}(v_i)$ indicates the selection of variant v_i and $\text{require}()$ indicates the require relationship. Similar notation will be used for rest of the rules definition. Due to this dependency rule the dependent 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.

4.3.2. Scenario 2

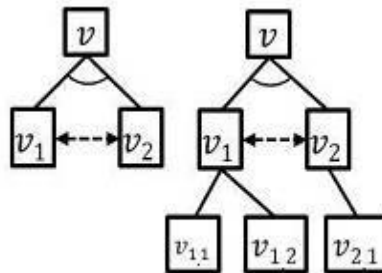


Fig 4.3.2: Exclude dependency between variants and between variation points

In Fig 5.3.2, 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 model well-formed. Similar to previous example, we define rules for such dependencies.

$$\forall v_1, v_2 \cdot \text{type}(v_1, \text{variant}) \wedge \text{type}(v_2, \text{variant}) \wedge \text{exclude_v_v}(v_1, v_2) \wedge \text{select}(v_1) \Rightarrow \text{notselect}(v_2)$$

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

4.3.3. Scenario 3

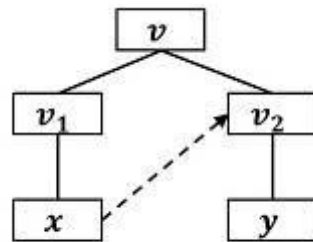


Fig 4.3.3: Require Dependency between variants and variation point

In Fig ,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

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

4.3.4 Scenario 4

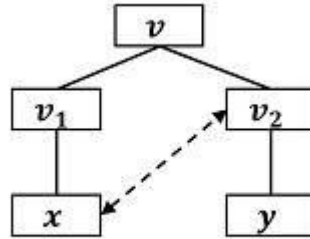


Figure 4.3.4.: Exclude dependency between variants and variation point

In Fig 4.3.4, 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 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. The following rule is derived from this scenario

$$\forall v_1, v_2, x, y \cdot \text{type}(x, \text{variant}) \wedge \text{type}(v_1, \text{variation point}) \\ \wedge \text{type}(v_2, \text{variation point}) \wedge \text{exclude } v_vp(x, v_2) \wedge \text{select}(x) \Rightarrow \text{notselect}(v_2)$$

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

4.3.5 Scenario 5

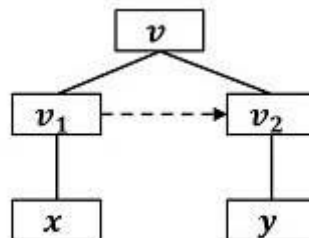


Figure 4.3.5.: Requires dependency between variation point

In Fig.4.3.5, 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

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

4.3.6 Scenario 6

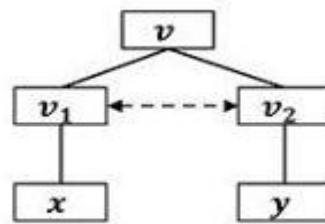


Figure 4.3.6: Exclude dependency between variation points

In Fig. 4.3.6, 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

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

4.3.7 Scenario 7

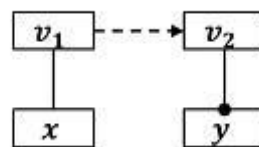


Figure 4.3.7: Exclude dependency between variation points

In Fig.4.3.7 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.

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

4.3.8 Scenario 8

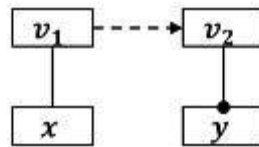


Figure 4.3.8: Variation point to variation point and parent-child relation

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

$$\forall v_1, v_2, x, y \cdot \text{type}(x, \text{variant}) \wedge \text{type}(y, \text{variant}) \wedge \text{variants}(v_1, x) \wedge \text{variant}(v_2, y) \wedge \text{common}(y) \wedge \text{requires_vp_vp}(v_1, v_2) \wedge \text{select}(x) \Rightarrow \text{select}(y)$$

4.3.9 Scenario 9

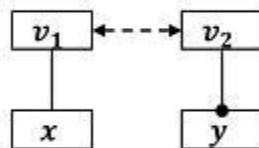


Fig 4.3.9: Variant and variation point exclude relation

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 often variant y , because we cannot select a variant when its variation point is not selected. The scenario is depicted in

$$\forall v_1, v_2, x, y : \text{type}(x, \text{variant}) \wedge \text{type}(y, \text{variant}) \wedge \text{variant}(v_1, x) \wedge \text{variant}(v_2, y) \wedge \text{common}(y, \text{yes}) \wedge \text{requires_vp_vp}(v_1, v_2) \wedge \text{select}(x) \Rightarrow \text{notselect}(y)$$

Chapter5

Model Verification in ALLOY

5.1 Representing our feature model using Alloy

Though the feature model that we analysis here is so vast so it would be very difficult to represent the whole feature model in Alloy. So we break the whole feature diagram into pieces. Then we try to encode each small pieces using alloy syntax. We assume that this small pieces give the correct result then after integrating all small pieces we can get the result about whether the feature model will work correctly or not. To do this first we take a small part of the GPL .

We can represent this part as below.

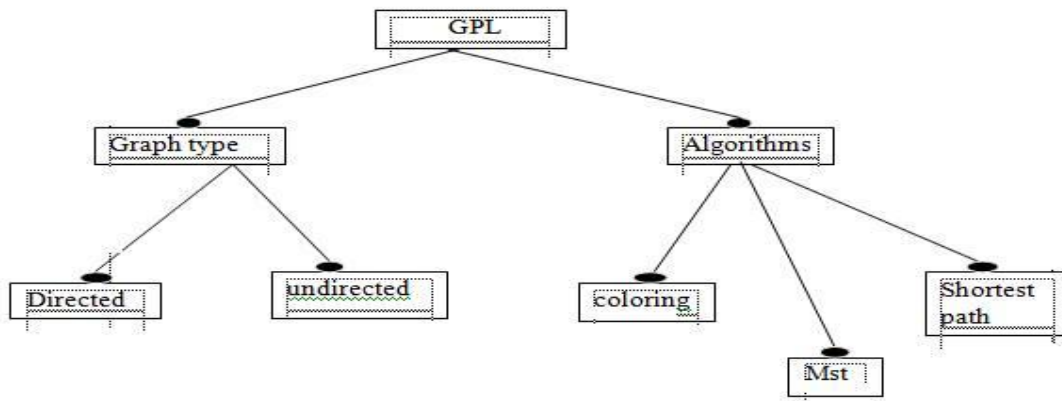


Figure :5.1.1 A small part of the GPL

In Alloy, one signature can extend another, establishing that the extended signature (sub-signature) is a subset of the parent signature. Firstly, we declare itselements; a singleton (one) sub-signature, which has exactly one object, for each FM element. The FM inis represented by GPL, which extends FM, and presents 2 features. A singleton signature is declared for each feature name. Finally we state GPL's features in a fact (fact), which packages formulas that always hold, such as invariants about the elements.

5.1.1 Alloy Encoding for above mention Tree

```
one sig CAD extends FM{}
one sig graph extends FM1{}
one sig directed,undirected extends Name{}
factCADFeatures
{
CAD.features1=graph
graph.features=directed+undirected
}
```

The “+” operator denotes the set union operator. Our main goal is to reason whether a transformation preserves or increases FM configurations. For that, FM semantics must be specified in Alloy. One approach is to declare an Alloy function yielding a set of valid configurations for a FM. By our concern in improving analysis performance, we cannot declare a semantics function for all FMs, which could be very inefficient. We then specified a semantics predicate for each FM. Part of this predicate is fixed for all FMs. The other part depends on its relationships and formulas. This encoding is systematic, straightforward for being included into tool support. Next, we explain the encoding through an example, later generalizing our approach. For each FM, a predicate is defined, containing all FM formulas directly translated to their semantics function. Using this approach, there is no predicate checking whether a configuration satisfies a formula. The immutable part of the semantics predicate introduce the following constraints: every configuration includes a subset of FMs names, and the root must always be included, as declared next. We call them implicit constraints.

5.1.2 Semantic Part of the above mentioned Tree

```
pred semantics graph[conf:set Name]
{
conf in graph.features
alternative[directed,undirected,conf]
}
```

Then all relationships of the FM are declared in terms of the predicates alternative [directed,undirected,conf] In order to systematically specify a FM into Alloy using our encoding, the following

steps must be taken::

- create a singleton sub-signature for each feature extending from Name
- specify the semantics predicate containing the relationships (reusing the encoding predicates) and formulas (using Alloy operators) in the FM.

Based on the previous encoding, we can perform automatic analysis on FMs using the Alloy Analyzer.

```
runsemanticsgraph
```

The run analysis command must specify a scope for all signatures declared. Our encoding contains 2 signatures. The previous fragment declares the run command for one FM (we are analyzing only one FM) and for 2 names (the FM encoded contains 4 features).

5.2 Alloy encoding

5.2.1 Output Validation for GPL

While checking for valid configuration we create a predicate valid configuration we create a predicate validconfig. where we select those feature that can product a valid product .Here we declare *graph, search, algorithm, directed, undirected, weighted, unweighted, DfsBfs, coloring, shortestpath, cycledetection, mst, stronglyconnected, approximation, brutef*. We have this option to select .So we need to select some of the features in order to get our desired product .Here our target is to check whether alloy gives us a valid result for our selection of features. In pred valid config we select *graph, algorithm, search, directed, weighted, Dfs, coloring, approximation*. We have already seen in the logic part that this combination gives us a valid result, which is indicated by the alloy result display screen ,where it shows that an instance is found. Which is shown in Fig-5.2.1 Also when we click on the 'instance' text then we get model which actually display the graph of the product and it is shown in fig5.2.1

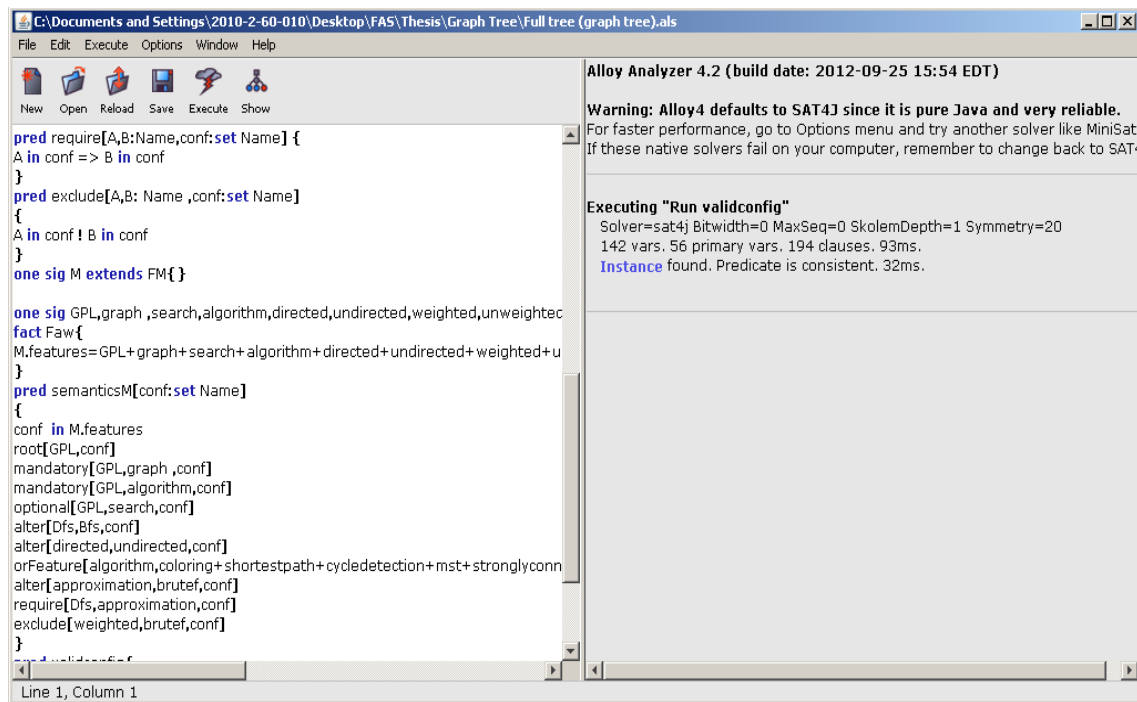


Fig:5.2.1 For validation for GPL

5.2.3 Output validation for hall booking system

While checking for valid configuration we create a predicate valid configuration we create a predicate validconfig. where we select those feature that can product a valid product. Here we declare *hallbooking, RCharge, RMode, Notification, RManagement, HandleConflicts, deposit, tax, basiccharge, discount, block, multiple rooms, multiple times, single, fax, print paper, email, add modify, delete*. We have this option to select. So we need to select some of the features in order to get our desired product. Here our target is to check whether alloy gives us a valid result for our selection of features. In pred valid config we select *hall booking, RCharge, RMode, Notification, RManagement, Handle Conflicts, deposit, block, multiple times, fax, addmodify, delete, add modify*. We have already seen in the logic part that this combination gives us a valid result, which is indicated by the alloy result display screen, where it shows that an instance is found. Which is shown in Fig-5.2.3 Also when we click on the 'instance' text then we get model which actually display the graph of the product and it is shown in fig 5.2.3

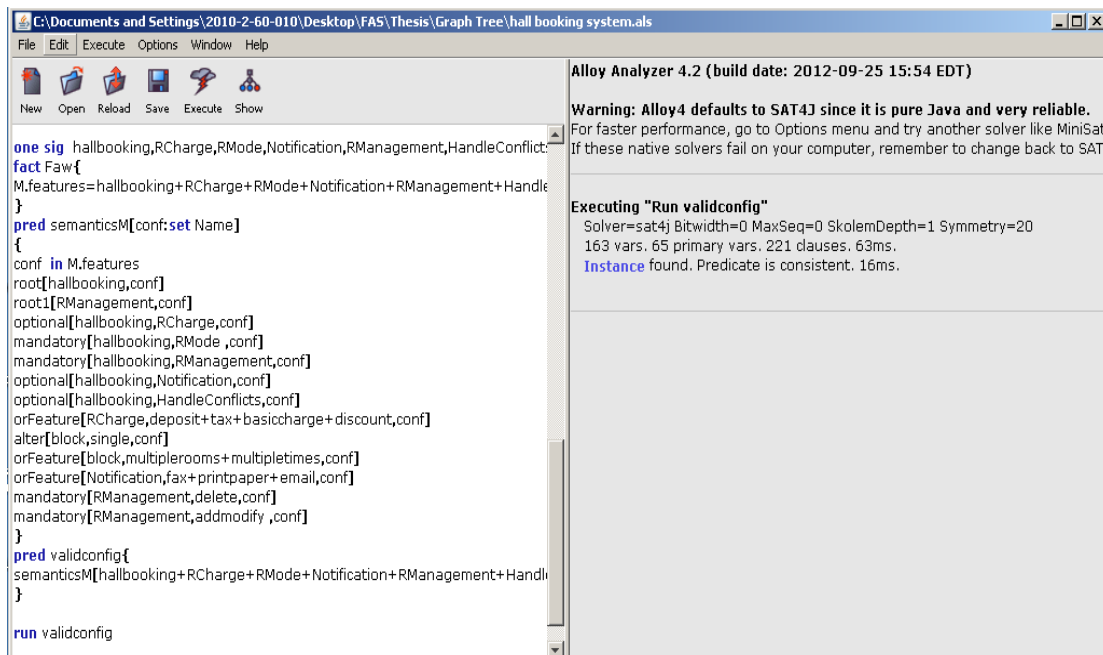


Fig:5.2.3 For validation of Hall Booking System

5.2.4 Output validation for Invalid configuration

Here is the Alloy (Fig 5.2.4) output for Invalid configuration for GPL feature tree

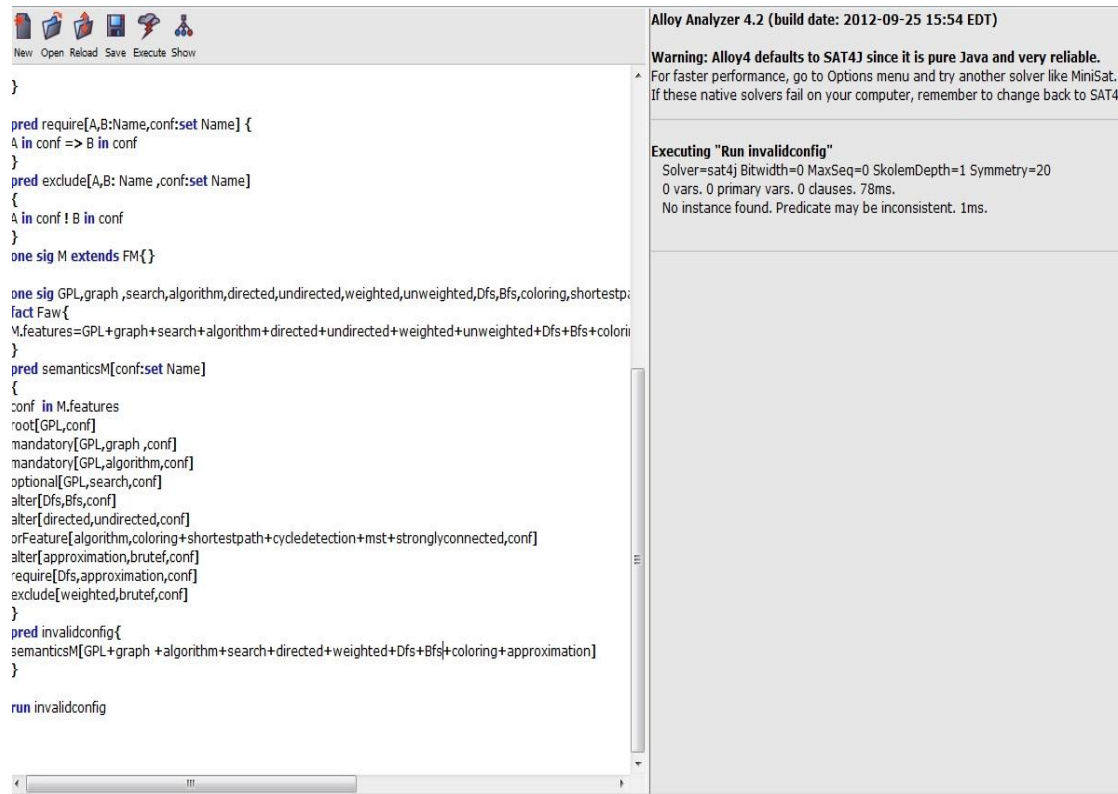


Fig 5.2.4 Run invalid Configfor GPL feature tree

Chapter 6

Conclusion

6.1 Conclusion

A product line architecture represents a significant long term investment .The ease with which the architecture can deal with changes such as- New features, new products and better quality properties will have a significant impact on the success of the product line. In a domain model, when the volume of information grows the possible explosion of variant combination becomes inevitable and tracing of proper variant information in the domain model becomes hard. As a result, the impacts on variants on domain model during customization of any particular product become unclear. A variant model has been proposed here which explicitly represents all the variant related information in order to generate any customized product form the domain model. Successful development of software product line requires appropriate organization and management of products requirements. A significant characteristic of developing product line is the management of the variants which is a crucial success factor of product line.

We presented an approach to formalizing and verifying SPL feature models to be able to create a decision table to generate customized product by using formal reasoning techniques. We provided formal semantics of the feature models by using, set representation first-order logic and specified the definitions of six types of variant relationships. We have six notations mandatory, optional, or, alternative, optional alternative, optional or We also defined cross-tree variant dependencies. Examples are provided describing various analysis operations, such as validity. We have addresses most of the analysis questions mentioned in . Finally, we encoded our logical notations into Alloy to be able to automatically verify any analysis related queries. A knowledge-based approach to specify and verify feature models is presented in . Comparing to that presentation, our definition relies on set representation which can be directly applied in many verification tools .

6.2 Future Work

Our particular interest is developing a tool to automatically generate customized product based on user requirement. In contrast to other automated analysis of feature model tools, e.g., at this stage, our tool is domain specific where these automated tools can be used as a supporting tool and can be used to automatically verify the derived product specification.

Appendix

A.1. Alloy encoding for GPL

```
sig FM{
features:set Name
}
sig Name{}
sigconf{}

pred optional[A,B:Name,conf:set Name] {
B in conf =>A in conf
}

pred mandatory[A,B:Name,conf:set Name]{
A in conf<=> B in conf
}

pred alter[A,B: Name ,conf:set Name]
{
A in conf ! B in conf
}

pred root[A:Name,conf:set Name]
{
A in conf
}

pred orFeature[A:Name, children:set Name, conf:set Name]
{
A in conf<=>some c:children|c in conf
#children>=1
}

pred require[A,B:Name,conf:set Name] {
A in conf => B in conf
}

pred exclude[A,B: Name ,conf:set Name]
```

```

{
A in conf ! B in conf
}

one sig M extends FM{
onesigGPL, graph, search, algorithm, directed, undirected, weighted, unweighted, Dfs, Bfs, coloring, shortestpath, cycledetection, mst, stronglyconnected, approximation, brutef extends
Name{}
factFaw{
M.features=GPL+graph+search+algorithm+directed+undirected+weighted+unweighted+Dfs+Bfs+coloring+shortestpath+mst+cycledetection+stronglyconnected+approximation+brutef
}
predsemanticsM[conf:set Name]
{
confinM.features
root[GPL, conf]
mandatory[GPL, graph , conf]
mandatory[GPL, algorithm, conf]
optional[GPL, search, conf]
alter[Dfs, Bfs, conf]
alter[directed, undirected, conf]
orFeature[algorithm, coloring+shortestpath+cycledetection+mst+stronglyconnected, conf]
alter[approximation, brutef, conf]
require[Dfs, approximation, conf]
exclude[weighted, brutef, conf]
}
predvalidconfig{
semanticsM[GPL+graph+algorithm+search+directed+weighted+Dfs+coloring+approximation]
}
Runvalidconfig

```

A.1.1 MetaModel for GPL

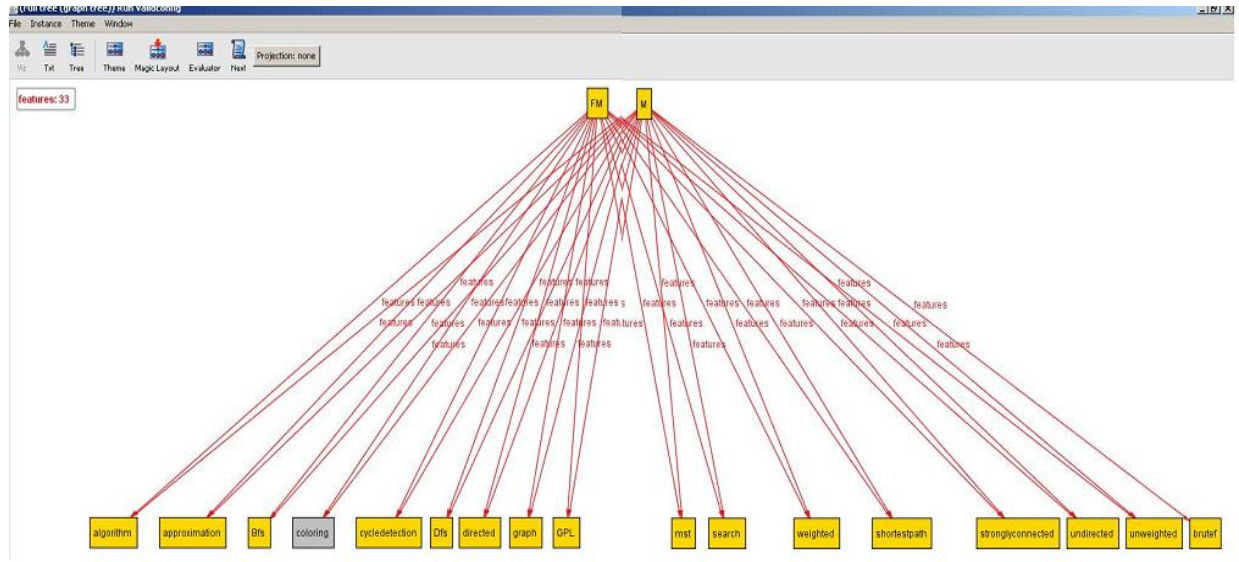


Fig: A.1.1 Metamodel of GPL Feature Tree

A.2. Alloy encoding for Hall booking system

```

sig FM{
  features:set Name
}
sig Name{}
sigconf{}
sig conf1{}
pred optional[A,B:Name,conf:set Name] {
  B in conf =>A in conf
}
pred mandatory[A,B:Name,conf:set Name]{
  A in conf<=> B in conf
}
pred alter[A,B: Name ,conf:set Name]
{
  A in conf ! B in conf
}
pred root[A:Name,conf:set Name]
{
  A in conf
}

```

```

}
pred root1[A:Name,conf:set Name]
{
A in conf
}
predorFeature[A:Name, children:set Name, conf:set Name]
{
A in conf<=>some c:children|c in conf
#children>=1
}
pred require[A,B:Name,conf:set Name] {
A in conf => B in conf
}
pred exclude[A,B: Name ,conf:set Name]
{
A in conf ! B in conf
}
one sig M extends FM{}
onesighallbooking,RCharge,RMode,Notification,RManagement,
HandleConflicts,deposit,tax,basiccharge,discount,block,mu
ltiplerooms,multipletimes,single,fax,printpaper,email,add
modify,delete extends Name{}
factFaw{
M.features=hallbooking+RCharge+RMode+Notification+RManage
ment+HandleConflicts+deposit+tax+basiccharge+discount+blo
ck+single+multiplerooms+multipletimes+single+fax+printpap
er+email+addmodify+delete
}
predsemanticsM[conf:set Name]
{
confinM.features
root[hallbooking,conf]
root1[RManagement,conf]
optional[hallbooking,RCharge,conf]
mandatory[hallbooking,RMode ,conf]
mandatory[hallbooking,RManagement,conf]
optional[hallbooking,Notification,conf]
optional[hallbooking,HandleConflicts,conf]
orFeature[RCharge,deposit+tax+basiccharge+discount,conf]
alter[block,single,conf]
orFeature[block,multiplerooms+multipletimes,conf]
orFeature[Notification,fax+printpaper+email,conf]
mandatory[RManagement,delete,conf]
mandatory[RManagement,addmodify ,conf]

```

```

}
predvalidconfig{
semanticsM[hallbooking+RCharge+RMode+Notification+RManage
ment+HandleConflicts+deposit+block+multipletimes+fax+adm
odify+delete+addmodify]
}

```

```
Runvalidconfig
```

A.2.1 Metamodel of Hall Booking System

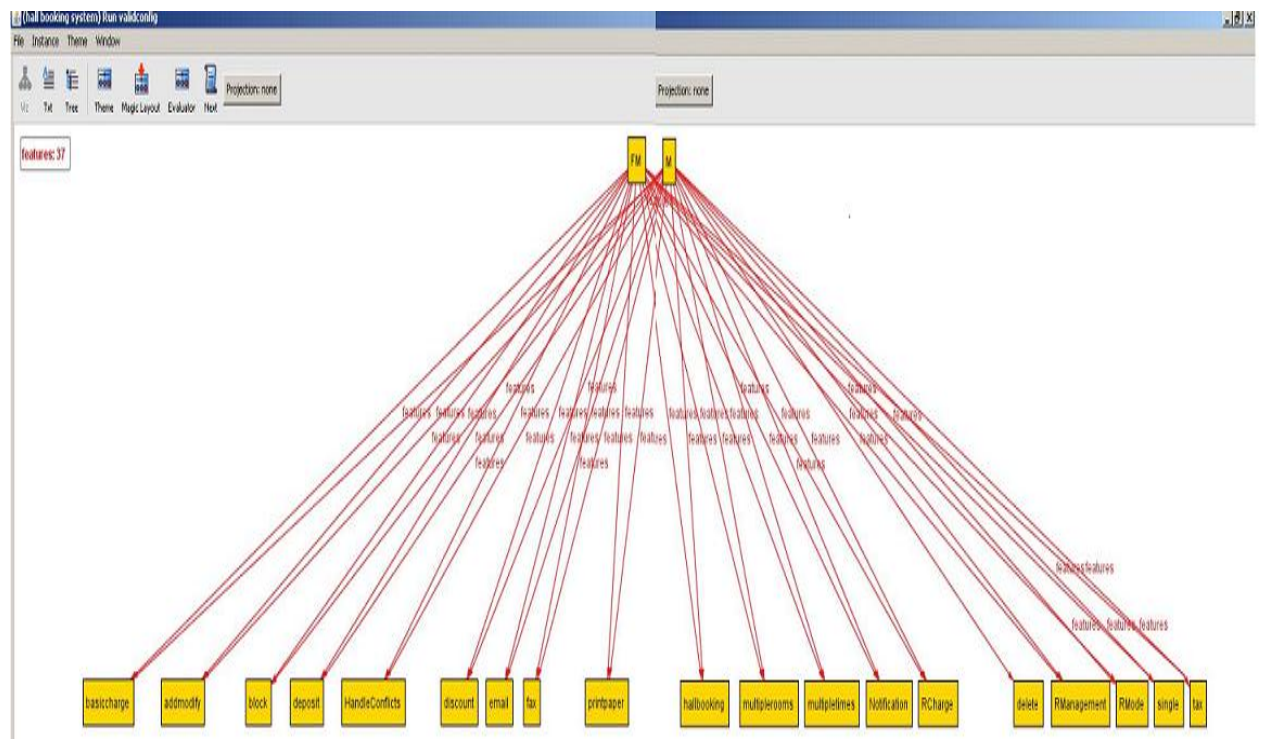


Fig:A.2.1 Metamodel of Hall Booking system Feature Tree

A.3. Invalid configuration for GPL

```

sig FM{
features:set Name
}
sig Name{}
sigconf{}

pred optional[A,B:Name,conf:set Name] {

```

```

B in conf =>A in conf
}

pred mandatory[A,B:Name,conf:set Name]{
A in conf<=> B in conf
}

pred alter[A,B: Name ,conf:set Name]
{
A in conf ! B in conf
}

pred root[A:Name,conf:set Name]
{
A in conf
}

predorFeature[A:Name, children:set Name, conf:set Name]
{
A in conf<=>some c:children|c in conf
#children>=1
}

predorfeature[A:Name, children:setName,conf:set Name]
{
A in conf<=> some c:children | c in conf
#children >1
}

pred alternative[A:Name, children: set Name, conf:set
Name]{
orFeature[A,children,conf]
#(children &conf) <=1
}

one sig M extends FM{}
onesigGPL,graph,search,algorithm,directed,undirected,weig
hted,unweighted,Dps,Bfs,spath,MST,coloring,approximation,
brutef extends Name{}
factFaw{
M.features=GPL+graph+search+algorithm+directed+undirected
+weighted+unweighted
+Dps+Bfs+spath+MST+coloring+approximation+brutef
}

```



```

predsemanticsM[conf:set Name]
{
  confinM.features
  root[GPL,conf]
  orfeature[GPL,MST,conf]
  orfeature[GPL,coloring,conf]
  alter[Dps,Bfs,conf]
  alter[directed,undirected,conf]
  alter[weighted,unweighted,conf]
  alter[approximation,brutef,conf]
  mandatory[GPL,graph ,conf]
  mandatory[GPL,algorithm,conf]
  optional[GPL,search,conf]

}
predInvalidconfig{
  semanticsM[GPL+
  graph+directed+brutef+algorithm+Dps+weighted+coloring+sea
  rch+MST+Bfs]
}
runInvalidconfig

```

Bibliography

- [1] DeBaud, J.M., Schmid, K.: A systematic approach to derive the scope of software product lines. In: Proceedings of the 21st International Conference on Software Engineering (ICSE), IEEE Computer Society Press (1999) 34–43
- [2] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
- [3] Bosch, J.: Design and Use of Software Architecture: Adopting and evolving a product-line approach. Addison-Wesley (2000)
- [4] Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004) To be published.
- [5] Deursen, A.v., Klint, P.: Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology* **10** (2002) 1–17
- [6] Griss, M., Favaro, J., d' Alessandro, M.: Integrating feature modeling with the RSEB. In: Proceedings of the Fifth International Conference on Software Reuse (ICSR), IEEE Computer Society Press (1998) 76–85
- [7] Lee, K., Kang, K.C., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In Gacek, C., ed.: Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr.15-19, 2002. LNCS 2319, Springer-Verlag (2002) 62–77
- [8] Barbeau, M., Bordeleau, F.: A protocol stack development tool using generative programming. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002. LNCS 2487, Springer-Verlag (2002) 93–109
- [9] Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002. LNCS 2487, Springer-Verlag (2002) 156–172
- [10] Riebisch, M., Bollert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with UML multiplicities. In: 6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA. (2002)
- [11] B. Chandrasekaran and John R. Josephson, Ohio State University V. Richard Benjamins, University of Amsterdam. “*What Are Ontologies, and Why Do We Need Them?*”
- [12] Peter F. Patel-Schneider, co-chair Bill Swartout, co-chair . “Description-Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing effort.” 1 November 1993

- [13] <http://www.racer-systems.com/products/racerpro/> (Last Visited in 11.11.2014)
- [14] [http://en.wikipedia.org/wiki/Propositional formula](http://en.wikipedia.org/wiki/Propositional_formula) (Last Visited in 11.11.2014)
- [15] Christian Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.
- [16] S. Jarzabek, Wai Chun Ong, and HongyuZhang. Handling variant requirements in domain modeling. *The Journal of Systems and Software*, 68(3):171–182, 2003.
- [17] <http://alloy.mit.edu/alloy/> (Last Visited in 17.11.2014)
- [19] <http://en.wikipedia.com> (Last Visited in 17.11.2014)
- [20] <http://wikipedia.org/wiki/wikipedia:citationneeded> (Last Visited in 17.11.2014)
- [21] *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [22] Andreas Hein, John MacGregor, and Steffen Thiel. Configuring software product line Features. In ElkePulvermller, Andreas Speck, James Coplien, Maja D Hondt, and Wolfgang De Meuter, editors, *Proceedings of the ECOOP 2001 Workshop on Feature Interaction in Composed Systems (FICS 2001), Budapest, Hungary, June 18-22, 2001*, volume 2001-14 of *Technical Report*, pages 67–69. University of Karlsruhe, Institutfür Programmstrukturen und Datenorganisation, 2001.
- [23] Jan Bosch. *Design and use of software architectures: adopting and evolving A product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [24] <http://www.sei.cmu.edu/productlines/> (Last Visited in 26.10.2014)
- [25] <http://www.tdgseville.info/topics/spl> (Last Visited in 26.10.2014)
- [26] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426, 2002.