Logic Based Verification of Software Product Line Feature Model

By

Md. Mosarrof Hossain

Department of Computer Science and Engineering
East West University

Summer 2016

East West University

Logic Based Verification of Software Product Line Feature Model

Submitted By

Md. Mosarrof Hossain
2011-2-60-008

Supervised By

Dr. Shamim H Ripon
Associate Professor
Department of Computer Science and Engineering

A thesis submitted in partial fulfillment for the degree of B.Sc in
Computer Science and Engineering
In the
Faculty of Science and Engineering
Department of Computer Science and Engineering

Summer 2016

Declaration

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

Md. Mosarrof Hossain

(2011-2-60-008)

Letter of Acceptance

The project entitled "**Logic Based Verification of Software Product Line Feature Model**" submitted by **Md. Mosarrof Hossain (ID: 2011-2-60-008)** 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 Summer 2016.

Board of Examiners

Dr. Shamim H Ripon
Associate Professor
Department of Computer Science and Engineering
East West University, Dhaka-1212, Bangladesh

Dr. Md. Mozammel Huq Azad Khan
Professor and Chairperson
Department of Computer Science and Engineering
East West University, Dhaka-1212, Bangladesh

Abstract

Formal verification of variant requirements has gained much interest in the software product line (SPL) community. Feature diagrams are widely used to model product line variants. However, there is a lack of precisely defined formal notation for representing and verifying such models. This report presents an approach to analyzing SPL variant feature diagrams using 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 a Computer Aided Dispatch (CAD) system variant feature model is presented to illustrate the analysis and verification process.

Acknowledgement

First of all Thanks to ALLAH for the uncountable blessings on us. Thanks to my Supervisor **Dr. Shamim H Ripon**, Associate Professor of Department of Computer Science and Engineering, East West University, for providing me this opportunity to test my skills in the best possible manner. He enlightened, encouraged and provided me with ingenuity to transform my vision into reality.

I am particularly grateful to **Dr. Md. Mozammel Huq Azad Khan**, Chairperson, Department of Computer Science and Engineering, East West University, for his encouragement, guidance and counseling.

Contents

List of Figures

List of Tables

To My Parents

And

All of My Teacher’s

CHAPTER 1
Introduction

1.1 Introduction

Designing, developing and maintaining a good software system is a challenge still in this 21st century. The approach of reusing existing good solutions for developing any new application is now one of the central focuses of software engineers. Building software systems from previously developed components saves cost and time of redundant work and improves the system and its maintainability. A new software development paradigm, software product line[1], is emerging to produce multiple systems by reusing the common assets across the systems in the product line. However, the idea of product line is not new. In 1976 Parnas[16] proposed modularization criteria and information hiding for handling product line.

The increase competitiveness in the software development sector with immense economic considerations such as cost, time to market, etc. motivates the transition from single product development to product-line development approach.

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 [1]. 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's of the products considered (development for reuse) and the derivation of product variants from these abstractions (development with reuse) [10].

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 vary among products in the family. This implies a huge model that helps 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 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 [6], 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.

- Our objectives are to logically representation of feature model facilitating the development of decision table in a formally sound way.

- A set of analysis operation has to be carried to out to check the consistency of the feature model. Our plan is to perform this verification by using our logical representation.

- It is often leveled that manual verification leads to numerous error for large models and often misses the minute details in the verification. It is our plan is to use a light-weight model checker to check the logical verification of the feature models.

In order to conduct out experiment we use a case study of Computer Aided Dispatch (CAD) system product line by analyzing and modeling the variants as well as the variants dependencies.

1.4 Contribution

The particular emphasis of this thesis is to model the variant of the product line in a manageable way so that product generation step can be conveniently handled. To achieve this goal in this thesis we have made the following contributions,

- We define six types of logical notation to represent all the parts in a feature model. First-order logic has been used for this purpose. These notations can be used to define all possible scenarios of a feature model.

- We then analyze the feature model using the logical definitions. After analyzing the feature model considering various scenarios, we define a set of rules which can be used to verify the feature model.

- The logical verifications are carried out by hand which is laborious task and error prone. To overcome this problem of our logical definitions we use the model checker Alloy[11]. Alloy use first order logic. We encode our logical definitions into Alloy and check the validity of the logical verification that we perform hand.

1.5 Outline

The thesis is organized as follows,

- Chapter 2 gives a brief overview of preliminary notations of propositional and first order logic is also presented in this chapter. We then give a brief review of the model checker Alloy with an example model.

- Chapter 3 gives an overview of the CAD domain. We briefly describe an example of a Police CAD domain. Then we present the logical representation of feature tree and analysis operation of feature tree. Also represent Alloy representation of the logical notations.

- Chapter 4 illustrates the steps how the logical representations are encoded into Alloy and how the verification has been performed.

- Chapter 5 concludes the thesis by summarizing our work. Finally we outline our future plan.

CHAPTER 2
Background

2.1 Software Product Line

A new software development paradigm, software product line [1], is emerging to produce multiple systems by reusing the common assets across the systems in the product line. However, the idea of product line is not new. In 1976 Parma's [16] proposed modularization criteria and information hiding for handling product line.

The increase competitiveness in the software development sector with immense economic considerations such as cost, time to market, etc. motivates the transition from single product development to product-line development approach. 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 missions and that are developed from a common set of core assets in a prescribed way [1].

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 [1]. 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's of the products considered (development for reuse) and the derivation of product variants from these abstractions (development with reuse)[10].

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 helps 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.

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 “¬”. Example:
p: This book is interesting.
Then P can be read as “This book is not interesting”.
Truth Table:

P	¬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¬P, is the proposition that is false when p is true and true when p is false.

2. Conjunction Operator: “and”, has symbol “∧”. Example
p: This book is interesting.
q: I am statying at home
p ∧ q: This book is interesting if and only if I am staying at home
Truth Table:

P	Q	P ∧ 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 is the proposition that is true when both p and q are true and false otherwise.

3. Disjunction Operator: inclusive "or", has symbol "∨". Example:
p: This book is interesting.
q: I am statying at home
p ∨ q: This book is interesting if and only if I am staying at home
Truth Table:

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 $p \vee q$. The disjunction p ∨ q of p and q is the proposition that is true when p is true, q is true, or both are true, and are false otherwise.

4. **Exclusive Or Operator:** "xor", has symbol⊕.Example:
p: This book is interesting.
q: I am statying at home
p ⊕ q: This book is interesting if and only if I am staying at home
Truth table:

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 $p \oplus q$, 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 statying at home
p ⟹ q: This book is interesting if and only if I am staying at home
Truth Table:

P	Q	P ⟹ Q
T	T	T
T	F	F
F	T	T
F	F	F

The implicationp $\Longrightarrow$ 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 statying at home
p $\Leftrightarrow$ q: This book is interesting if and only if I am staying at home.
Truth table:

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

The bi-conditional statement is equivalent to$(p \Longrightarrow q) \wedge (q \Longrightarrow p)$.In other words: For p $\Leftrightarrow$ q to be true we must have both p and q true.

2.3 Alloy

The Alloy Analyzer is a software tool which can be used to analyze specifications written in the Alloy specification language. The Analyzer can generate instances of model invariants; simulate the execution of operations defined as part of the model, and check user-specified properties of a model. The Alloy Analyzer supports the analysis of partial models. As a result, it can perform incremental analysis of models as they are constructed, and provide immediate feedback to users. The simulations performed by the Alloy Analyzer tool are sound and complete up to a given scope. If there is some instance that contradicts an assertion up to a given scope, the tool shows a counterexample. However, if the tool does not find any counterexample, we only know that the property holds on that scope.

Alloy and Alloy Analyzer were developed by Daniel Jackson's group at MIT

- 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.

2.3.1 Syntax and Semantics

To explain the semantics here an object model for alloy tree is given in Fig. 2.1.

Each box denotes a set of objects. In Alloy these are called signatures. In Alloy sets of atoms such as Man, Woman, Married, Person are called signatures. Textual representation starts with sig declarations defining the signatures. Signatures correspond to object classes. A signature that is not subset of another signature is a top-level signature. Here Person and Name are top-level signatures. Extensions of a signature are also disjoint. Man andWoman are disjoint sets. An abstract signature has no elements except those belonging to its extensions. There is no Person who is not a Man or a Woman.

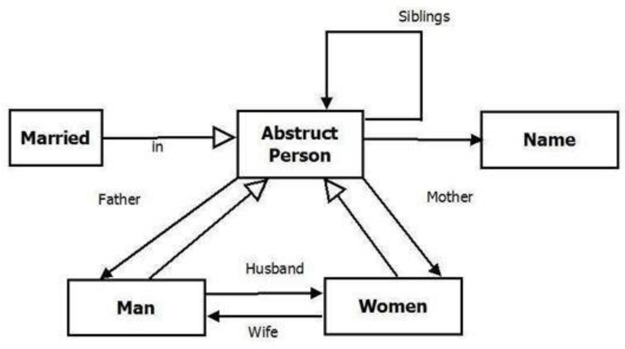


Figure 2.1: Family Diagram

Arrows with a small filled arrow head denote relations. For example, name is a relation that maps Person to Name. 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 one), some (one or more), extends and in are used to denote which signature is subset of which other signature. For instance:

```
sig A {}                     // Set of Atoms A
sig B {}                     //Set B is a subset of A
sig B extends A {}           // B and C are disjoint subsets of A: B in
sig B extends A {}           // A && C in A && no B & C
sig C extends A {}
abstract sig A {}            // A partitioned by disjoint subsets B
sig B extends A {}           //and C: no B && C && A = (B+C)
sig C extends A {}
one sig A {}                 // A is singleton set
lone sig B {}                // B is a singleton or empty
some sig C {}                // C is non empty set
```

The fields define relations among the signatures. Visual representation of a field is an arrow with a small filled arrow head (⟶). For example sig A {f: e}, where f is a binary relation with domain A and range given by expression e and each element of A is associated with exactly one element from e. After the signatures and their fields, facts are used to express constraints that are assumed to always hold. Facts are not assertions; they are constraints that restrict the model. For example

```
sig Host {}
sig Link {from, to: Host}
fact {
all x: Link | x.from != x.to
}// no links from a host to itself
```

A **function** is a named expression with zero or more arguments. When it is used, the arguments are replaced with the instantiating expressions. Syntax of writing a function is **fun** f[x1: e1... xn: en]: e {E}

A **predicate** is a named constraint with zero or more arguments. It is defining by using the keyword "**pred**".

In Alloy, assertions are used to specify properties about the specification. Assertions state the properties that we expect to hold.

In order to perform an analysis we need to use run command based on a predicate.

Now we are giving a simple example named "Checking Own Grandpa" using alloy. The alloy code is given below

```
Module language/Family
sig Name { }
abstract sig Person {
name: one Name,
siblings: Person,
father: lone Man,
mother: lone Woman
  }
sig Man extends Person {
wife: lone Woman
  }
sig Woman extends Person{
husband: lone Man
  }
sig Married extends Person {
  }
fact {
  no p: Person | p in p.^(mother + father)
  wife = ~husband
}
fun grandpas[p: Person] : set Person {
set parent = mother + father +father.wife+mother.husband |
p.parent.parent& Man
  }
predownGrandpa[p: Person] {
p in grandpas[p]
  }
runownGrandpa for 4 Person
```

The corresponding graph generated by Alloy is shown in Figure 2.2.

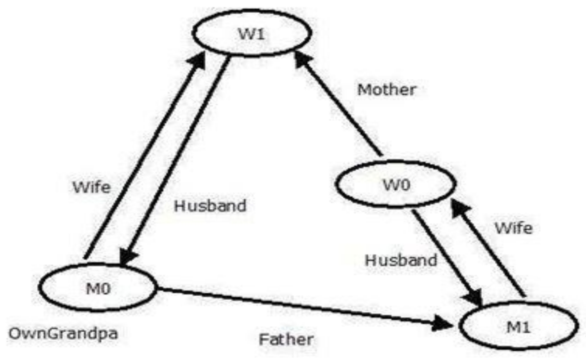


Figure 2.2 checking own grandpa

CHAPTER 3
Feature Model and Logical Representation

Our case study is based on the Computer Aided Dispatch System (CAD) domain. We consider this domain as our case study because we have got supporting documents of this domain from the company who is working on this domain and also working in collaboration with our research team. Several research works have already been done on different aspects of this domain which help to gain better knowledge of this domain. An overview of the Computer Aided Dispatch System (CAD) and its basic domain model is presented in this chapter.

3.1 Overview of CAD Domain

A Computer Aided Dispatch system (CAD) is a mission-critical system that is used by police, fire and rescue, health service, port operation, taxi booking and others. However, the basic operational scenarios are similar in all the CAD systems. Figure 3.1 depicts a basic operational scenario and roles in a CAD system.

When an incident is happened in a place a Caller reports the incident to the command and control centre of the police unit. A Call Taker in the command and control center captures the details about the incident and the Caller, and creates a task for the incident. There is a Dispatcher in the system whose task is to dispatch resources to handle any incident. The system shows the Dispatcher a list of un-dispatched task.

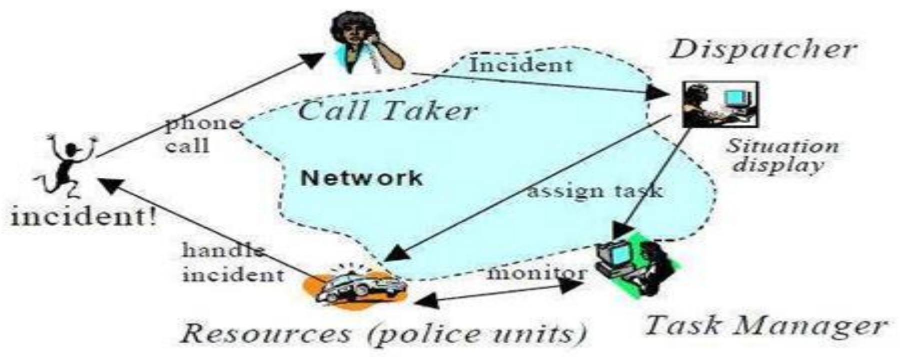


Figure 3.1 : Basic operational scenario in a CAD system for police

The Dispatcher examines the situation, selects suitable Resources (e.g. police units) and dispatches them to execute the task. The Resources carry out the task instructions and report to the Task Manager. The Task Manager monitors the situation and at the end when the resources finished the task- closes the task. Different CAD members have different resources and tasks for their system. The key entities of CAD domain will interact with each other according to the system requirements. For example, some resources are free of charge (e.g. police) whereas some are not (e.g. taxi). The basic key entities of CAD domain are listed in Table 3.1.

Table 3.1: Key entities of CAD domain

Key	Definitions
Task	A task holds together the information regarding a particular incident (incident location, type, priority, urgency, status, caller’s detail etc).Two separate entities are included within it incident’s detail and caller detail.
Resources	A resource handles the task. In police CAD, the resource would be police car, in taxi CAD, the resource would be taxi. it is presumed that the resource are equipped with necessary hardware to communicate with CAD system
Command	Command instructs the resources to complete a task. The person who is responsible for dispatching task to resources sends the command.

At the basic operational level, all CAD systems are similar; basically they support the dispatcher units to handle the incidents. However, there are differences across the CAD systems. The specific context of operation results in many variations on the basic operational theme. Here are some of the variants identified in CAD domain: then call taker informs the dispatcher of the newly created task but if merged then without informing to dispatcher he/she can dispatch resources directly to the incidents.

Validation of caller and task information differs across CAD systems. In some CAD systems basic validation (i.e., checking the completeness of caller information and the task information) is sufficient while in other CAD systems validation includes duplicate task checking, etc. in yet other CAD systems no validation is required at all.

Un-dispatched task selection rule in certain situation at any given time there might be more than one task to be dispatched, then there is a need to decide which task will dispatched next. A number of algorithms are available for this purpose and different CAD system use different algorithm. In Ambulance CAD system task may be selected based on task urgency or priority whereas in taxi system different algorithm will be applied.

This simple description of CAD variants hints us about numerous variants and variant dependencies, which focus the importance of managing them properly.

3.2 CAD Domain Model using FODA

Modeling variants is an important process during designing software product line. The feature oriented domain analysis (FODA) method was developed at the Software Engineering Institute (SEI) [13].FODA focuses on identifying features that characterize a domain. Features are user visible aspects or characteristics of a system and are organized into and/or graph in order to identify the commonalities and variants of the application domain. Feature modeling is an integral part of the FODA method and the Feature Oriented Domain Reuse Method (FORM) [14]. The commonalities and variants within features are exploited to create a set of models that is used to implement any member product of that family.

Features are represented in graphical form as trees. The internal nodes of a tree represent the variants and their leaves represent values of corresponding variants. Graphical symbols are used to indicate the categories of features. The root node of a feature tree always represents the domain whose features are modeled. The remaining nodes represent features which are classified into three types:

- Mandatory features are always part of the system.

- Optional features may be selected as a part of the system if their parent feature is in the system. The decision whether an optional feature is part of the system or not can be made independently from the selection of other features.

- Alternative features of a variant are related to each other as exclusive-or relationship, i.e. exactly one feature out of a set of features is to be selected.

The feature diagram depicts the classification of mandatory features and variant features as well as their dependencies. Mandatory features are those which are present in all products in the respective domain. Variant features appear only some members of the domain which differentiate one product from others. There are more relationships between features. One is Or-feature by [8], which connects a set of optional features with a parent feature, either common or variant. The meaning is that whenever the parent feature is selected then at least one of the optional features will be selected. Feature diagram also depicts the interdependencies among the variants which describes the selection of one variant depends on the selection of the dependency connected variants. A partial CAD feature diagram is given in the Figure. In this feature diagram the root represents the functional features of CAD. Task Assignment Rule, Call Taker & Dispatcher Roles and Validation are linked to the root via mandatory link as these are mandatory features of CAD. However, Checking Duplicate Task is linked via optional link as this feature is optional. We use extensions described in [8].

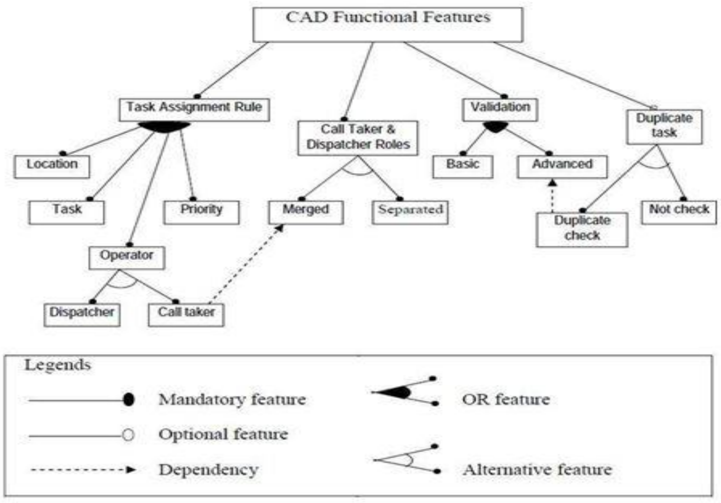


Figure 3.2 : Partial Feature Diagram of a CAD system

3.3 Logical Representation

3.3.1 Introduction

Logic representation involves analysis of how to reason accurately, effectively represent a set of facts using a set of symbols within a knowledge domain. A symbol vocabulary and a system of logic are combined to enable inferences of a particular domain. Logic usually has a well-defined syntax, semantics and proof theory.

- The syntax of logic defines the syntactically acceptable objects of the logic.
- The semantics of logic associates each formula with a meaning.
- The proof theory is concerned with manipulating formulae according to certain rules.

Logic is used to supply formal semantics of how reasoning functions should be applied to the symbols in the System domain. Logic is also used to define how operators can process and reshape the knowledge.

There are two types of logic that we can apply in our work in order verify CAD(Computer Aided Dispatch) System .They are Propositional logic and First Order Logic(FOL). But we decide to use FOL instead of Propositional logic be-cause of some reason. They are as below

- It is possible to add lemmas to FOL provers, which can make future proof easier.
- The translation of property languages into FOL is inherently more flexible and

modular. Propositional logic translation are more flexible but not as compact, due to the restrictiveness of this language.

- In FOL methods it is possible to verify properties of the generic designs. Propositional logic methods can only verify specific instances.

Propositional logic, also known as statement logic, is the branch of logic that studies ways of joining or modifying entire propositions, statements or sentences.

Also used for representing logical relationships and properties those are derived from these methods of combining or altering statements. In propositional logic, the simplest statements are considered as indivisible units. The most thoroughly researched branch of propositional logic is classical truth-functional propositional logic, which studies logical operators and connectives that are used to produce complex statements whose truth-value depends entirely on the truth-values of the simpler statements making them up, and in which it is assumed that every statement is either true or false. It is concerned with propositions and their inter-relationships. A proposition is a possible condition of the world about which we want to say something. In Propositional Logic, there are two types of sentences – Simple sentences and compound sentences. Simple sentences express “atomic” propositions about the world. Compound sentences express logical relationships between the simpler sentences of which they are composed. propositional logic does not consider smaller parts of statements, and treats simple statements as in-divisible wholes, the language PL uses uppercase letters ‘A’, ‘B’, ‘C’, etc., in place of complete statements. The logical signs ‘$\wedge$’, ‘$\vee$’, ‘$\longrightarrow$’, ‘$\longleftrightarrow$’, and ‘$\neg$’ are used in place of the truth-functional operators, “and”, “or”, “if then”, “if and only if”, and ’not’, respectively.

First-order logic is symbolized reasoning in which each sentence, or statement, is broken down into a subject and a predicate. In first-order logic, a predicate can only refer to a single subject. First-order logic is also known as first-order predicate calculus or first-order functional calculus. A sentence in first-order logic is written in the form P (x), where P is the predicate and x is the subject, represented as a variable. Syntax used in First Order Logic: Basic elements Constants

- variables denoted by x, y, z, v, u, . . .,
- constants denoted by a, b, c, d, . . .,
- function symbols denoted by f, g, . . .,
- relation symbols denoted by p, q, r, . . ., or P, Q, R, . . .,
- propositional constants, which are true and false,
- connectives, which are$\neg$ (negation),$\bigvee$(disjunction),$\bigwedge$(conjunction),
  $\Longrightarrow$ (Implication) and $\Leftrightarrow$ (equivalence)
- quantifiers, which are $\exists$ (there exists) and $\forall$(for all).

Propositional logic assumes the world contains facts, first-order logic assumes the world contains

- Objects: people, houses, numbers
- Relations: bigger than, part of.
- Functions: one more than.

3.3.2 Logic Representation of Feature Model

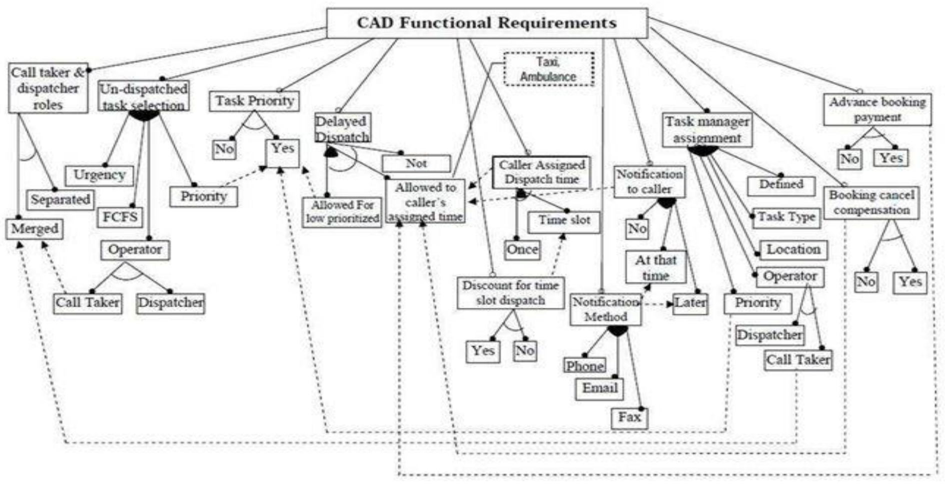


Figure 3.3: CAD feature diagram

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. Every Feature is an increment in product functionality. The complete feature tree of CAD domain is illustrated in Fig 3.3. It can be used in deferent 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 features X must also be included”. The relation between a parent (variation point) features and its child features (variants) are categorized as follows:

Mandatory: A child feature is said to be mandatory when it is required to appearwhen 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 theparent features appears. For instance, it is optional to have pdf reader software in the mobile phone.

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

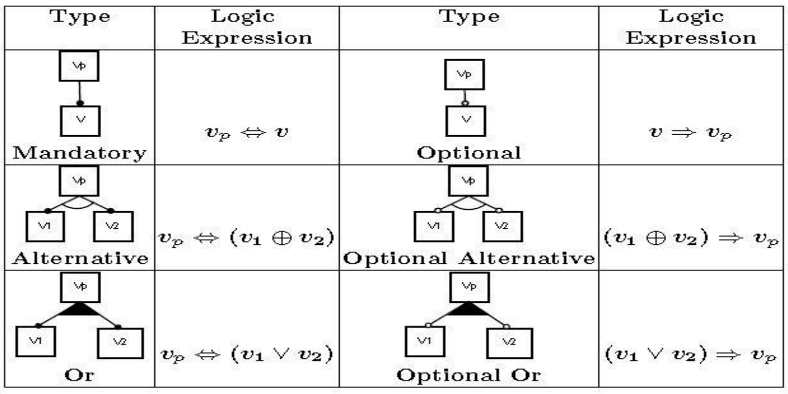
Optional Alternative: One feature from a set of alternative features may or maynot be included if parent included.

Or: A set of child features are said to have an Or-relation with their parent whenone 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.

The logical notions of these features are defined in the following Figure 3.4

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 vi) and the variants (vi. j) by using the expressions shown in Fig. 4. 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 3.4 : logical notation for feature tree

3.4 Analysis Operations

Our target performs some operation that tells us feature model works correctly or not. That means some time feature model contain inconsistency that give us wrong product. So we have to give certain opportunities to identify those problems and also keep user from selecting those wrong selections of feature. Our proposed method also supports some extra feature like inconsistency, dead feature, false optional etc.

3.4.1 Inconsistency

Inconsistency in a feature model is a relationship between features that cannot be true at the same time. To explain more about it Fig, is given above which contain inconsistency. In figure, $v, v1, v2$ are three variation points. Where $v1$ contain two variants$v1.1$ and $v1.2$. Both the variants are mandatory. That means whenever the variation point $v1$ is selected then both the variants $v1.1$ and $v1$will be automatically selected by the system. Variation point $v2$ contains two variants $v2.1$ and $v2.2$.

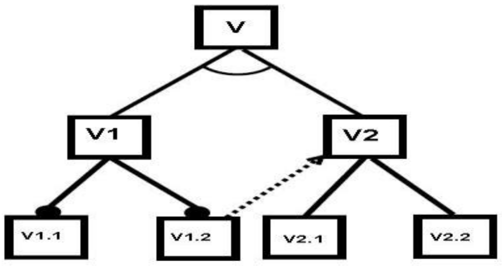


Figure 3.5 : Inconsistency checking

Also there exists a require relationship between variant $v1.2$ and variation point $v2$. That means whenever $v1.2$ is selected then variation point $v2$ must be selected by the system. But it can't be possible because both the variation point $v1$ and $v2$ are connected with their variation point$v$ with an alternative relationship. This means we cannot select both $v1$ and $v2$ at a time.That's why system reports an error. We can define a rule for this situation

$\forall v1, v2, v1.2\colon type(v1, variantionpoint) \wedge type(v2, variantionpoint)$
$\wedge type(v1.2, variant) \wedge variants(v, v1{,}2) \wedge exclude\_vp\_v(v1.2, v2) \wedge$
$select(v1.2) \Rightarrow error$

3.4.2 False Optional

False optional is situations where some feature are declared as optional which does not need to be declared. Because those features are automatically selected by the system when some other feature are selected by the user. This is actually waste of time. Figure, depicts how this type of situation occurs in a system. In Figure 3.6 $v, v1, v2$ are three variation points. Where $v1$ is a mandatory feature of the variation point is $v\ and\ v2$ is an optional feature of the variation point$v$. Also $v1$ contain two variants $v1.1$ and $v1.2$. Both the variants are mandatory. That means whenever the variation point $v1$ is selected then both the variants $v1.1$ and $v1.2$ will be automatically selected by the system. Variation point $v2$ contains two variants $v2.1$ and $v2.2$. Also there exists a require relationship between variant $v1.2$ and variation point $v2$.

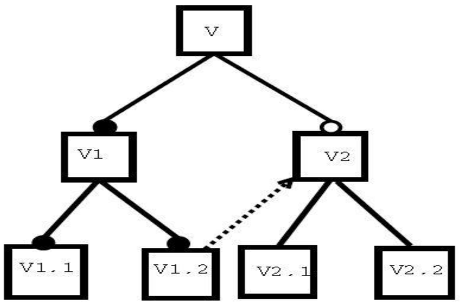


Figure 3.6: False Optional checking

That means selection of the feature $v1.2$ will require the selection of $v2$. On other way we can say that whenever we select either the variation point $v1$ or any variants of $v1$ then $v2$ will be automatically selected by the system. Actually $v2$ acted like a mandatory feature of$v$. So we shouldn't need to declare this variation point as an optional feature. The following rules are defined for such cases

$\forall v1, v2, v1.2{:}\ type(v1, variantion\ point) \wedge type(v2, variantion\ point)$
$\wedge type(v1.2, variant) \wedge require\_v\_vp(v1.2, v2) \wedge select(v1.2) \Rightarrow select(v2)$

$\forall v1, v2, v1.2{:}\ type(v1, variantion\ point) \wedge type(v2, variantion\ point)$
$\wedge type(v1.2, variant) \wedge variants(v, v1.2) \wedge exclude\_v\_vp(v1.2, v2) \wedge$
$select(v1.2) \Rightarrow notselect(v2)$

3.4.3 Dead Feature Detection

A dead feature is a feature that never appears in any legal product of feature model and sometime it causes error in feature model. To show how dead feature can occur in a feature model Fig 3.7 , is given below. In Fig 3.7,v , $v1, v2$ are three variation point. Where $v1$ is a mandatory feature of the variation point v and $v2$ is also a mandatory feature of the variation point$v$. Also $v1$ contain two variants $v1.1$ and $v1.2$. Both the variants are mandatory .That means whenever the variation point $v1$ is selected then both the variants $v1.1$ and $v1.2$ will be automatically selected by the system.

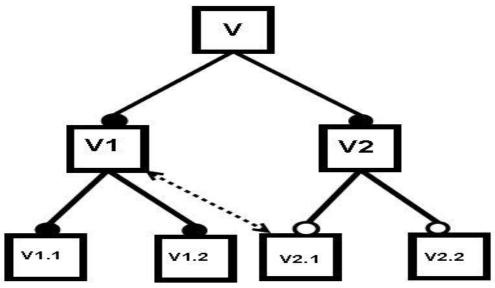


Figure 3.7: Dead Feature Detection

Variation point $v2$ contains two variants $v2.1$and $v2.2$, which are optional feature of $v2$. Also there exists an exclude relationship between variant $v2.1$ and variation point $v1$. That means variation point $v1$ and variant $v2.1$ will never appear in same product. Whenever any feature of$v1$ is selected then $v1$ is automatically selected which restrict a user from selecting the variant $v1.2$ .though $v1$ is a mandatory feature of the system that means $v1$ will appear in every product, so $v2.1$ will never appear in any product although variation point of the variant $v2.1$ is also mandatory feature of the system. We can only select the variant $v2.2$. so $v2.1$ will be a dead feature.

$\forall v1, v2, v2.1, v1.1: type(v1, variantion\ point) \wedge type(v2, variantion\ point)$
$\wedge type(v1.1, variant) \wedge type(v2.1, variant) \wedge variants(v1.1, v1) \wedge$
$variants(v2.1, v2) \wedge excludes\_v\_vp(v2.1, v1) \wedge select(v1.1) \Longrightarrow$
$dead\_feature(v2.1)$

3.5 Example

Automatic analysis of variants is already identified as a critical task. Various operations of variant analysis are suggested in [7][8]. Our logical representation can define and validate a number of such analysis operations. The validation of a product line model is assisted by its logical representation. While constructing a single system from a product line model, we assign TRUE (T) value to selected variants and FALSE (F) to those not selected. After substituting these values to product line model, if TRUE value is evaluated, we call the model as valid otherwise the model is invalid. A product graph is considered to be valid if the mandatory sub-graphs are evaluated to TRUE. It’s very difficult to represent the whole feature tree so we take a partial feature tree for our analysis and it is shown In Fig 3.8.

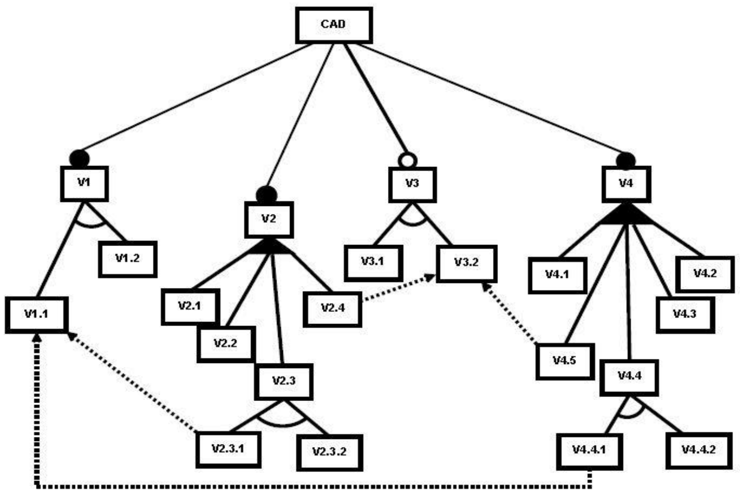


Figure 3.8: Partial CAD feature Tree

First we break the entire graph shown in figure into four small sub-graph and named those sub-graph as G1, G2, G3 and G4 respectively. In this sub-graph we used short notation instead of using feature names showed in the original CAD feature diagram. Description of all the sub-graph is given below.

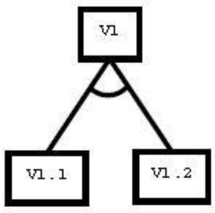


Figure 3.9: G1 Sub graph

Fig 3.9 shows the diagram of G1 sub-graph. Here we use $v1, v1.1$ and $v1.2$ instead of "Call taker and dispatcher roles", "Merged", "Separated" respectively. Here in G1 sub-graph $v1$ is denoted as a variation point and $v1.1$ and $v1.2$ are denoted as variants, which belong to the variation point $v1$. Both the variants $v1.1$ and $v1.2$ are mutually exclusive to each other that mean we cannot select both at the same time.

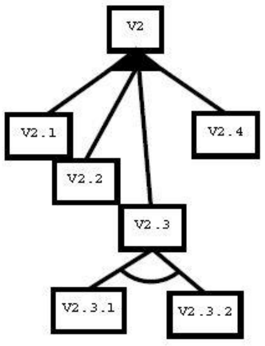


Figure 3.10 G2 sub Graph

In G2 sub graph figure 3.10 here we use $v2, v2.1, v2.2, v2.3, v2.3.1, v2.3.2$ and $v4$ instead of "Un-dispatched task selection", "Urgency", "FCFS", "Operator", "Call Taker", "Dispatcher", and "Priority". Here in G2 sub-graph $v2$ is denoted as a variation point and $v2.1$, $v2.2$, $v2.3$, $v2.3.1$, $v2.3.2$, $v2.4$ are denoted as variants, which belongs to the variation point $v2$. But later $v2.3$ is denoted a variation point, having the variants $v2.3.1$ and $v2.3.2$. Here in this sub-graph $v2.1$, $v2.2$, $v2.3$, $v2.4$ are connected with its variation point with Or relationship, which means anyone can select one or more variants at a time. Also $v2.3.1$ and$v2.3.2$ variants are connected with its variation point $v2.3$ with an alternative relationship. There exists two require relationship between the variants $v2.3.1$ and $v2.1$ and variants $v2.4$ and $v3.2$.Which indicates that you cannot select $v2.3.1$ unless you select $v2.1$ and you cannot select $v2.4$ unless you select $v3.2$.

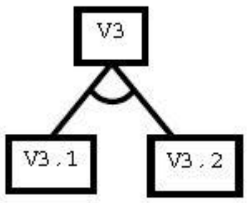


Figure 3.11:G3 sub graph

In Fig 3.11 shows the diagram of G3 sub-graph. Here we use $v3$, $v3.1$ and $v3.2$ instead of “Task Priority”, “No”, “Yes”. In G3 sub-graph $v3$ is denoted as variation point and $v3.1, v3.2$ are denoted as variants, which belongs to the variation point $v3$. Both the variants $v3.1$ and $v3.2$ are mutually exclusive to each other. That means we cannot select both at a time.

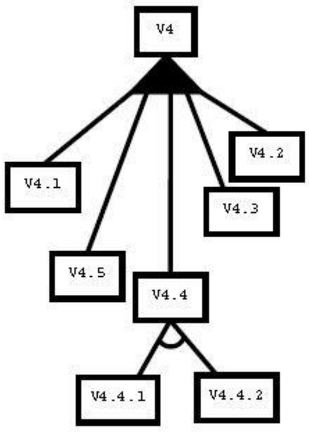


Figure 3.12: G4 Sub graph

In G4 sub Graph figure 3.12 Here we use $v4$, $v4.1$, $v4.2$, $v4.3$, $v4.4$, $v4.4.1$, $v4.4.2$, and $v4.5$ instead of “Task manager assignment”, “Defined”, “Task Type”, “Location”, “Operator”, “Dispatcher”, “Call Taker”, “Priority”. Here in G4 sub-graph $v4$ is denoted as a variation point and $v4.1$, $v4.2, v4.3$, $v4.4$, $v4.4.1, v4.4.2$, $v4.5$ are denoted as variants, which belongs to the variation point $v4$. But later $v4.4$is denoted as a variation point, having the variants $v4.4$ and $v4.4.2$. Here in this sub-graph $v4.1$, $v4.2$, $v4.3, v4.4$, $v4.5$ are connected with its variation point with Or relationship, which means anyone can select one or more variants at a time. Also $v4.4.1$ and $v4.4.2$ variants are connected with its variation point $v4.4$ with an alternative relationship. There exists two require relationship between the variants $v4.4.2$ and $v1.1$ and variants $v4.5$ and $v3.2$.Which indicates that you cannot select $v4.4.2$ unless you select $v1.1$ and you cannot select $v4.5$ unless you select $v3.2$.

3.5.1 Example 1

Suppose the selected variants are $v1,\ v1.2,\ v2,\ v2.3,\ v2.3.1,\ v2.4,\ v3,\ v3.2,\ v4,\ v4.1,\ v4.4,\ v4.4.2$ and $v4.5$. Now our work is to check this input combination can build a valid product. To do this First we have to check whether each individual sub-graph is valid or not, then the dependency associate with them and at last we have to check whether the whole graph can build a valid product. Here we check the validity of the sub-graph G1, G2, G3 and G4 by substituting the truth values of the variants of the sub-graphs

G1:
$(v1.1 \oplus v1.2)$
$= (T \oplus F) \Leftrightarrow T$
$= T \Leftrightarrow T$
$= T$
Here $v1.1$ and $v1.2$ are mutually exclusive to each other so we use an XOR ($\oplus$) notation between them and also only that time G1 is True when both $v1$ and any one from $v1.1$ and $v1.2$ is true. Here we have seen that $v1$ and $v1.1$ is selected. Mathematical analysis given above tells us that G1 is valid for this input combination.

G2:
$(v2.1 \vee v2.2 \vee v2.3 \vee v2.4) \Leftrightarrow v2$
$= \big(v2.1 \vee v2.2 \vee \big((v2.3.1 \oplus v2.3.2) \Leftrightarrow v2.3\big) \vee v2.4\big) \Leftrightarrow v2$
$= \big(T \vee F \vee \big((T \oplus F) \Leftrightarrow T\big) \vee T\big) \Leftrightarrow T$
$= (T \vee F \vee (T \Leftrightarrow T) \vee T) \Leftrightarrow T$
$= (T \Leftrightarrow T)$
$= T$

Here variant $v2.1,\ v2.2,\ v2.3,\ v2.4$ are connected with their variation point $v2$ with Or ($\vee$) relationship, that means we can select one or more feature. In this scenario G2 will valid when at least one of the variant from $v2.1, v2.2, v2.3$ and $v2.4$ will be true. Also variation point $v2.3$ contains two variants $v2.3.1$ and $v2.3.2$. Here a required dependency exists between $v2.3.1$ and variants $v1.1$ that means anyone cannot select $v2.3.1$ unless $v1.1$ is selected. Another requires dependency also exist between the variant $v2.4$ and $v3.2$. Here also anyone cannot select $v2.4$ unless he selects $v3.2$. Mathematical analysis described above tells us that G2 is valid for this input.

G3:
$(v3.1 \oplus v3.2) \Leftrightarrow v3$
$= (F \oplus T) \Leftrightarrow T$
$= T \Leftrightarrow T$
$= T$

Here $v3.1$ and $v3.2$ are mutually exclusive to each other so we use an XOR ($\oplus$) notation between them and also only that time G3 is True when both $v3$ and any one from $v3.1$ and $v3.2$ is true. Here we have seen that $v3$ and $v3.2$ is selected. Mathematical analysis given above tells us that G3 is valid for this input combination.

G4:
$(v4.1 \vee v4.2 \vee v4.3 \vee v4.4 \ \vee V4.5) \Leftrightarrow v4$
$= \big(v4.1 \vee v4.2 \vee v4.3 \vee \big((v4.4.1 \oplus v4.4.2) \Leftrightarrow v4.4\big) \vee v4.5\big) \Leftrightarrow v4$
$= \big(T \vee F \vee F \vee \big((F \oplus T) \Leftrightarrow T\big) \vee T\big) \Leftrightarrow T$
$= (T \vee F \vee F \vee (T \Leftrightarrow T) \vee T) \Leftrightarrow T$
$= (T \vee F \vee F \vee T \vee T) \Leftrightarrow T$
$= T \Leftrightarrow T$
$= T$

Here variant $v4.1,\ v4.2,\ v4.3,\ v4.4,\ v4.5$ are connected with their variation point $V4$ with Or ($\vee$) relationship, which means we can select one or more feature. In this scenario G4 will valid when at least one of the variant from $v4.1,\ v4.2,\ v4.3,\ v4.4,$ $v4.5$ will be true. Also variation point $v4.4$ contains two variants $v4.4.1, v4.4.2$. Here a required dependency exists between $v4.4.1$ and variants $v1.1$ that means anyone cannot select $v4.4.1$ unless $v1.1$ is selected. Another requires dependency also exist between the variant $v4.5$ and $v3.2$. Here also anyone cannot select $v4.5$ unless he selects $v3.2$. Mathematical analysis described above tells us that G4 is valid for this input combination.

As the sub-graph G1, G2, G3 and G4 are evaluate to TRUE, the product model is valid. However, variant dependencies are not considered in this case. Dependencies among variants are defined as additional constraints which must be checked separately apart from checking the validity of the sub-graphs. Evaluating the dependencies of the selected variants, we get,

Dependency:

$(v2.3.1 \Longrightarrow v11) \wedge (v2.4 \Longrightarrow v3.1) \wedge (v4.4.2 \Longrightarrow v1.1) \wedge (v4.5 \Longrightarrow v3.1)$
$= T \wedge T \wedge T \wedge T$
$= F$

Though all the sub-graphs are valid and also dependency among the variant are valid so it concludes that the selected features from the feature model create a valid product.

3.5.2 Example 2

Suppose the selected variants are $v1,\ v1.2,\ v2,\ v2.3,\ v2.3.1,\ v2.4,\ v3,\ v3.2,\ v4,$ $v4.1,\ v4.4,\ v4.4.2$ and $v4.5$. Now our work is to check this input combination can build a valid product. To do this First we have to check whether each individual sub-graph is valid or not, then the dependency associate with them and at last we have to check whether the whole graph can build a valid product. Here we check the validity of the sub-graph G1, G2, G3 and G4 by substituting the truth values of the variants of the sub-graphs.

G1:

$(v1.1 \oplus v1.2)$
$= (F \oplus T) \Leftrightarrow T$
$= T \Leftrightarrow T$
$= T$

Here $v1.1$ and $v1.2$ are mutually exclusive to each other so we use an XOR ($\oplus$) notation between them and also only that time G1 is True when both $v1$ and any one from $v1.1$ and $v1.2$ is true. Here we have seen that $v1$ and $v1.1$ is selected. Mathematical analysis given above tells us that G1 is valid for this input combination.

G2:

$(v2.1 \vee v2.2 \vee v2.3 \vee v2.4) \Leftrightarrow v2$
$= \big(v2.1 \vee v2.2 \vee \big((v2.3.1 \oplus v2.3.2) \Leftrightarrow v2.3\big) \vee v2.4\big) \Leftrightarrow v2$
$= \big(F \vee T \vee \big((F \oplus T) \Leftrightarrow T\big) \vee T\big) \Leftrightarrow T$
$= (F \vee T \vee (T \Leftrightarrow T) \vee T) \Leftrightarrow T$
$= (T \Leftrightarrow T)$
$= T$

Here variant $v2.1,\ v2.2,\ v2.3,\ v2.4$ are connected with their variation point $v2$ with Or (∨) relationship, that means we can select one or more feature. In this scenario G2 will valid when at least one of the variant from $v2.1,\ v2.2,\ v2.3$ and $v2.4$ will be true. Also variation point $v2.3$ contains two variants $v2.3.1$and$v2.3.2$. Here a required dependency exists between $v2.3.1$ and variants $v1.1$ that means anyone cannot select $v2.3.1$ unless $v1.1$ is selected. Another requires dependency also exist between the variant $v2.4$ and $v3.2$. Here also anyone cannot select $v2.4$ unless he selects $v3.2$. Mathematical analysis described above tells us that G2 is valid for this input combination.

G3:

$(v3.1 \oplus v3.2) \Leftrightarrow v3$
$= (T \oplus F) \Leftrightarrow T$
$= T \Leftrightarrow T$
$= T$

Here$v3.1$ and $v3.2$ are mutually exclusive to each other so we use an XOR ($\oplus$) notation between them and also only that time G3 is True when both $v3$ and any one from $v3.1$ and $v3.2$ is true. Here we have seen that $v3$ and $v3.2$ is selected. Mathematical analysis given above tells us that G3 is valid for this input combination.

G4:

$$
\begin{aligned}
&(v4.1 \vee v4.2 \vee v4.3 \vee v4.4\ \vee V4.5) \Leftrightarrow v4 \\
&= \big(v4.1 \vee v4.2 \vee v4.3 \vee \big((v4.4.1 \oplus v4.4.2) \Leftrightarrow v4.4\big) \vee v4.5\big) \Leftrightarrow v4 \\
&= \big(T \vee F \vee F \vee \big((T \oplus F) \Leftrightarrow T\big) \vee T\big) \Leftrightarrow T \\
&= (T \vee F \vee F \vee (T \Leftrightarrow T) \vee T) \Leftrightarrow T \\
&= (T \vee F \vee F \vee T \vee T) \Leftrightarrow T \\
&= T \Leftrightarrow T \\
&= T
\end{aligned}
$$

Here variant $v4.1,\ v4.2,\ v4.3,\ v4.4,\ v4.5$ are connected with their variation point $V4$ with Or ($\vee$) relationship, which means we can select one or more feature. In this scenario G4 will valid when at least one of the variant from $v4.1,\ v4.2,\ v4.3,\ v4.4,$ $v4.5$ will be true. Also variation point $v4.4$ contains two variants $v4.4.1$ and $v4.4.2$. Here a required dependency exists between $v4.4.1$ and variants $v1.1$ that means anyone cannot select $v4.4.1$ unless $v1.1$ is selected. Another requires dependency also exist between the variant $v4.5$ and $v3.2.$ Here also anyone cannot select $v4.5$ unless he selects $v3.2$. Mathematical analysis described above tells us that G4 is valid for this input combination.

As the sub-graph G1, G2, G3 and G4 are evaluate to TRUE, the product model is valid. However, variant dependencies are not considered in this case. Dependencies among variants are defined as additional constraints which must be checked separately apart from checking the validity of the sub-graphs. Evaluating the dependencies of the selected variants, we get,

Dependency:

$$
\begin{aligned}
&(v2.3.1 \Rightarrow v11) \wedge (v2.4 \Rightarrow v3.1) \wedge (v4.4.2 \Rightarrow v1.1) \wedge (v4.5 \Rightarrow v3.1) \\
&= T \wedge F \wedge T \wedge F \\
&= F
\end{aligned}
$$

Here we have seen that for the current input combination all the sub-graph gives valid result but we do not get a valid result for dependency. When we define dependency we have seen that $v2.4$ require the selection of variant $v3.2$ but here $v3.1$ is selected so ($v2.4 => v3.1$) gives false result. Also ($v4.5 => v3.1$) gives false result. Though all the dependencies between variants are connected through AND ($\wedge$) operator. So we get false value while analyzing dependency. So it is impossible to construct a valid Product.

CHAPTER 4
Alloy Verification

4.1 Representing our feature model using Alloy

Though the feature models 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 piece 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 CAD system. We can represent this part as below.

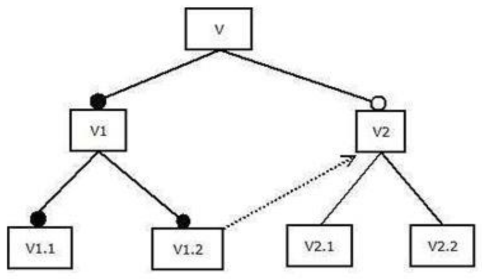


Figure 4.1: Small part of CAD system

In Alloy, one signature can extend another, establishing that the extended signature (sub signature) is a subset of the parent signature. Firstly, we declare its elements; a singleton (one) sub signature, which has exactly one object, for each FM element.

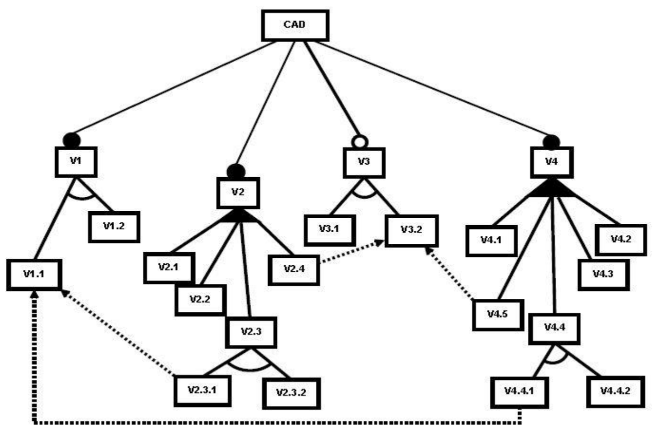


Figure 4.2: Partial CAD system feature tree

The FM in Figure 4.1 is represented by CAD, which extends FM, and presents two features. A singleton signature is declared for each feature name. Finally we state CAD's features in a fact (fact), which packages formulas that always hold, such as invariants about the elements.

```
one sig CAD extends
FM{} one sig v1
extends FM1{}
one sig v1.1,v1.2 extends Name{}
factCADFeatures {
CAD.features1=v1
v1.features=v1.1+v1.2
}
```

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

```
pred semanticsV1[conf:set Name] {
conf  in v1.features
alternative[v1.1,v1.2,conf]
}
```

Then all relationships of the FM are declared in terms of the predicates alternative [v1.1, v1.2, 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. Figure 4.1 has exactly one FM and some feature names. Since it is known the exact number of objects for all signatures (FM and Name) in our encoding, we can perform a complete analysis using the Alloy Analyzer in the FM of Figure 4.1.

```
run semanticsV1
```

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).

4.2 Alloy Encoding

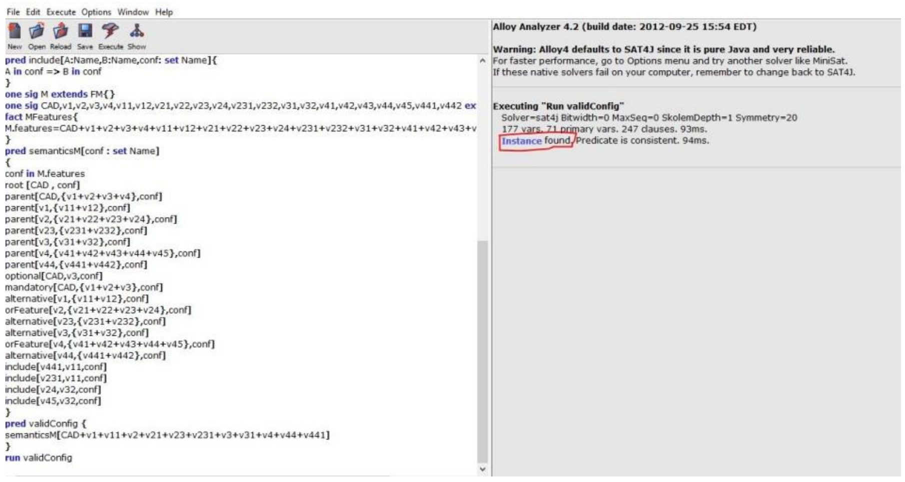
4.2.1 Checking Valid Configuration

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

! (A in B)
}
predorFeature[A:Name, children:setName,conf: set Name]{
A in conf<=> some c: children | c in conf
#children >1
}
pred alternative[A:Name,children:setName,conf:set Name]{
orFeature[A,children,conf]
# (children &conf) <=1
}
pred include[A:Name,B:Name,conf: set Name]{
A in conf => B in conf
}
one sig M extends FM{}
one sig CAD,v1,v2,v3,v4,v11,v12,v21,v22,v23,v24,v231,v43,
v232, v31, v41, v42, v44, v45, v441, v442 extends Name {}
factMFeatures{
M.features=CAD+v1+v2+v3+v4+v11+v12+v21+v22+v23+v24+v231+
v232+v31+v32+v41+v42+v43+v44+v45+v441+v442
}
predsemanticsM[conf : set Name]
{
conf in M.features
root [CAD , conf]
parent[CAD,{v1+v2+v3+v4},conf]
parent[v1,{v11+v12},conf]
parent[v2,{v21+v22+v23+v24},conf]
parent[v23,{v231+v232},conf]
parent[v3,{v31+v32},conf]
parent[v4,{v41+v42+v43+v44+v45},conf]
parent[v44,{v441+v442},conf]
optional[CAD,v3,conf]
mandotary[CAD,{v1+v2+v3},conf]
alternative[v1,{v11+v12},conf]
orFeature[v2,{v21+v22+v23+v24},conf]
alternative[v23,{v231+v232},conf]
alternative[v3,{v31+v32},conf]
orFeature[v4,{v41+v42+v43+v44+v45},conf]
alternative[v44,{v441+v442},conf]
include[v441,v11,conf]
include[v231,v11,conf]
include[v24,v32,conf]
include[v45,v32,conf]
}
predvalidConfig {
semanticsM[CAD+v1+v11+v2+v21+v23+v231+v3+v31+v4+v44+v441]
}
runvalidConfig
```

Output Meta Model

While checking for valid configuration we create a predicate validConfig. We selects $v1, v2, v3, v4, v11, v21,\ v22,\ v23,\ v231,\ v232,\ v24,\ v31,\ v32,\ v41,\ v42,\ v43,$ $v44,\ v45,\ v441, v442$ Instead of using our feature of feature tree. Instead of using our feature of feature tree. 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 predValidConfig we select $CAD,\ v1,\ v11,\ v2,\ v21,\ v23\ ,\ v231,\ v3, v31,\ v4,\ v44,\ v441$ .we have already seen in the logic part that this combination gives us a valid result. So when we run this predicate we get a valid result, which is indicated by the alloy result display screen, where it shows that an instance is found. Which is shown in Figure 4.3.Also when we click on the ”instance” text then we get a meta model which actually display the graph of our feature model and it is shown in Figure 4.4.


File Edit Execute Options Window Help
New Open Reload Save Execute Show

```
pred include[A:Name,B:Name,conf: set Name]{
A in conf => B in conf
}
one sig M extends FM{}
one sig CAD,v1,v2,v3,v4,v11,v12,v21,v22,v23,v24,v231,v232,v31,v32,v41,v42,v43,v44,v45,v441,v442 ex
fact MFeatures{
M.features=CAD+v1+v2+v3+v4+v11+v12+v21+v22+v23+v24+v231+v232+v31+v32+v41+v42+v43+v
}
pred semanticsM[conf : set Name]
{
conf in M.features
root [CAD , conf]
parent[CAD,{v1+v2+v3+v4},conf]
parent[v1,{v11+v12},conf]
parent[v2,{v21+v22+v23+v24},conf]
parent[v23,{v231+v232},conf]
parent[v3,{v31+v32},conf]
parent[v4,{v41+v42+v43+v44+v45},conf]
parent[v44,{v441+v442},conf]
optional[CAD,v3,conf]
mandatory[CAD,{v1+v2+v3},conf]
alternative[v1,{v11+v12},conf]
orFeature[v2,{v21+v22+v23+v24},conf]
alternative[v23,{v231+v232},conf]
alternative[v3,{v31+v32},conf]
orFeature[v4,{v41+v42+v43+v44+v45},conf]
alternative[v44,{v441+v442},conf]
include[v441,v11,conf]
include[v231,v11,conf]
include[v24,v32,conf]
include[v45,v32,conf]
}
pred validConfig {
semanticsM[CAD+v1+v11+v2+v21+v23+v231+v3+v31+v4+v44+v441]
}
run validConfig
```

```
Alloy Analyzer 4.2 (build date: 2012-09-25 15:54 EDT)

Warning: Alloy4 defaults to SAT4J since it is pure Java and very reliable.
For faster performance, go to Options menu and try another solver like MiniSat.
If these native solvers fail on your computer, remember to change back to SAT4J.

Executing "Run validConfig"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   177 vars. 71 primary vars. 247 clauses. 93ms.
   Instance found. Predicate is consistent. 94ms.
```

Figure 4.3: Check for Validity

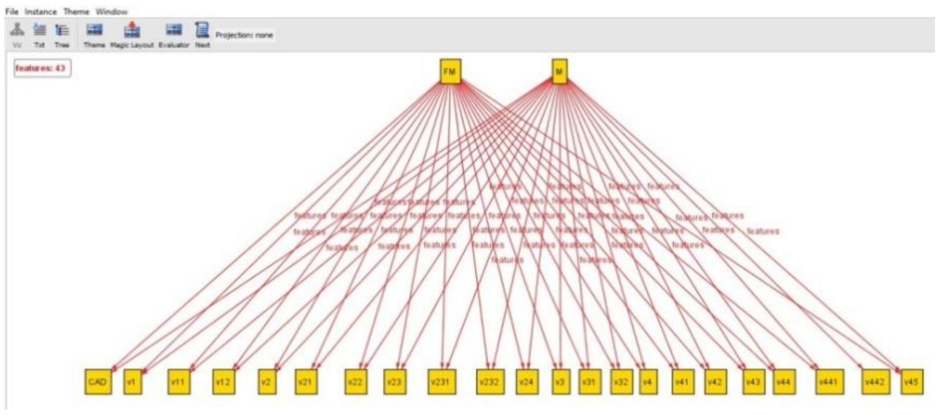


Figure 4.4: Meta model of valid configurations

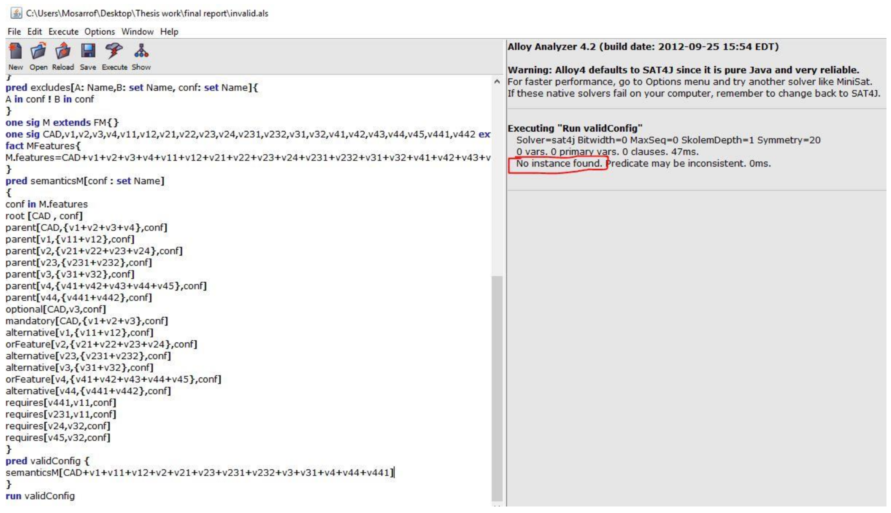
4.2.2 Checking Invalid Configuration

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

```
pred optional[A:Name,B:setName,conf: set Name] {
B in conf =>A in conf
}
pred mandatory[A:Name,B:setName,conf:set Name] {
A in conf<=> B in conf
}
pred alter[A:Name,B:setName,conf:set Name]{
A in conf! B in conf
}
pred root[A:Name,conf:set Name] {
A in conf
}
pred parent[A:Name,B:setName,conf:set Name]{

!(A in B)
}
predorFeature[A:Name,children:setName,conf: set Name]{
A in conf<=> some c: children | c in conf
#children >1
}
pred alternative[A:Name,children:setName,conf:set Name]{
orFeature[A,children,conf]
#(children &conf) <=1
}
pred include[A:Name,B:Name,conf: set Name]{
A in conf => B in conf
}
one sig M extends FM{}
one sig CAD,v1,v2,v3,v4,v11,v12,v21,v22,v23,v24,v231,v43,
v232,  v31,  v32,  v41,  v42,  v44,  v45,  v441,  v442  extends
Name {}
factMFeatures{
M.features=CAD+v1+v2+v3+v4+v11+v12+v21+v22+v23+v24+v231+
v232+v31+v32+v41+v42+v43+v44+v45+v441+v442
}
predsemanticsM[conf :set Name]
{
conf in M.features
root [CAD,conf]
parent[CAD,{v1+v2+v3+v4},conf]
parent[v1,{v11+v12},conf]
parent[v2,{v21+v22+v23+v24},conf]
parent[v23,{v231+v232},conf]
parent[v3,{v31+v32},conf]
parent[v4,{v41+v42+v43+v44+v45},conf]
parent[v44,{v441+v442},conf]
optional[CAD,v3,conf]
mandotary[CAD,{v1+v2+v3},conf]
alternative[v1,{v11+v12},conf]
orFeature[v2,{v21+v22+v23+v24},conf]
alternative[v23,{v231+v232},conf]
alternative[v3,{v31+v32},conf]
orFeature[v4,{v41+v42+v43+v44+v45},conf]
alternative[v44,{v441+v442},conf]
include[v441,v11,conf]
include[v231,v11,conf]
include[v24,v32,conf]
include[v45,v32,conf]
}
predNotvalidConfig {
semanticsM[CAD+v1+v11+v12+v2+v21+v23+v231+v232+v3+v31+v4+
v44+v441]
}
runNotvalidConfig
```

While checking for valid configuration we create a predicate validConfig. We selects $v1, v2, v3, v4, v11, v12, v21$, $v22$, $v23$, $v231$, $v232$, $v24$, $v31$, $v32$, $v41$, $v42$, $v43$, $v44$, $v45$, $v441, v442$ Instead of using our feature of feature tree. We have these option to select. So we need to select some of these 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 NotvalidConfig we selectCAD, $v1$, $v11, v12$, $v2$, $v21$, $v23$, $v231$, $v3$, $v31$, $v4$, $v44$, $v441$. we has already seen in the logic part that this combination gives us a not valid result. So when we run this predicate we get a not valid result, which is indicated in the alloy result display screen, where it shows that an instance is not found. This is shown in Figure 4.5. Here though we reach to our destination, but still we have lots of things to do. Our future work using alloy will be the calculation of the number of valid product that can be constructed also though alloy is completely text based so if we can build a graphical interface and also connected with the alloy analyzer, then it will be easier for people to check any feature model created by them.


C:\Users\Mosarrof\Desktop\Thesis work\final report\invalid.als
File Edit Execute Options Window Help
New Open Reload Save Execute Show

```
}
pred excludes[A: Name,B: set Name, conf: set Name]{
A in conf ! B in conf
}
one sig M extends FM{}
one sig CAD,v1,v2,v3,v4,v11,v12,v21,v22,v23,v24,v231,v232,v31,v32,v41,v42,v43,v44,v45,v441,v442 ex
fact MFeatures{
M.features=CAD+v1+v2+v3+v4+v11+v12+v21+v22+v23+v24+v231+v232+v31+v32+v41+v42+v43+v
}
pred semanticsM[conf : set Name]
{
conf in M.features
root [CAD , conf]
parent[CAD,{v1+v2+v3+v4},conf]
parent[v1,{v11+v12},conf]
parent[v2,{v21+v22+v23+v24},conf]
parent[v23,{v231+v232},conf]
parent[v3,{v31+v32},conf]
parent[v4,{v41+v42+v43+v44+v45},conf]
parent[v44,{v441+v442},conf]
optional[CAD,v3,conf]
mandatory[CAD,{v1+v2+v3},conf]
alternative[v1,{v11+v12},conf]
orFeature[v2,{v21+v22+v23+v24},conf]
alternative[v23,{v231+v232},conf]
alternative[v3,{v31+v32},conf]
orFeature[v4,{v41+v42+v43+v44+v45},conf]
alternative[v44,{v441+v442},conf]
requires[v441,v11,conf]
requires[v231,v11,conf]
requires[v24,v32,conf]
requires[v45,v32,conf]
}
pred validConfig {
semanticsM[CAD+v1+v11+v12+v2+v21+v23+v231+v232+v3+v31+v4+v44+v441]
}
run validConfig
```

```
Alloy Analyzer 4.2 (build date: 2012-09-25 15:54 EDT)

Warning: Alloy4 defaults to SAT4J since it is pure Java and very reliable.
For faster performance, go to Options menu and try another solver like MiniSat.
If these native solvers fail on your computer, remember to change back to SAT4J.

Executing "Run validConfig"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   0 vars. 0 primary vars. 0 clauses. 47ms.
   No instance found. Predicate may be inconsistent. 0ms.
```

Figure 4.5: Check for error

CHAPTER 5
Conclusion

5.1 Summary

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 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 first-order logic and specified the definitions of six types of variant relationships. We also defined cross-tree variant dependencies. Examples are provided describing various analysis operations, such as validity, inconsistency, dead feature detection etc. We have addresses most of the analysis questions mentioned in [3, 4]. 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 [9]. Comparing to that presentation, our definition relies on first-order logic which can be directly applied in many verification tools as in [19]. In contrast to other approaches [2, 5, 7,12, 15, 21], our proposed method defines across-graph variant dependencies as well as dependencies between variation point and variants.

5.2 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. [20], 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. We are also planning to model the UML class diagrams of the domain added with variants.

BIBLIOGRAPHY

[1] *Software product lines: practices and patterns*. Addison-Wesley LongmanPublishing Co., Inc., Boston, MA, USA, 2001.

[2] Don S. Batory. Feature models, grammars, and propositional formulas. In J. HenkObbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[3] David Benavides, Antonio Ruiz Cort´es, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feature models. In *JISBD*, pages 367–376, 2006.

[4] David Benavides, Sergio Segura, and Antonio Ruiz Cort´es. Automated analy-sis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615– 636, 2010.

[5] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cort´es. Automated rea-soning on feature models. In *Proceedings of the 17th international conferenceon Advanced Information Systems Engineering*, CAiSE'05, pages 491–503,Berlin, Heidelberg, 2005. Springer-Verlag.

[6] Jan Bosch. *Design and use of software architectures: adopting and evolvinga product-line approach*. ACM Press/Addison-Wesley Publishing Co., NewYork, NY, USA, 2000.

[7] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glck and Michael R. Lowry, editors, *GPCE*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

[8] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming:methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co.,New York, NY, USA, 2000.

[9] Abdelrahman Osman Elfaki, SomnukPhon-Amnuaisuk, and Chin Kuan Ho. Knowledge based method to validate feature models. In Steff en Thiel and Klaus Pohl, editors, *SPLC (2)*, pages 217–225. Lero Int. Science Centre, Uni-versity of Limerick, Ireland, 2008

[10] Andreas Hein, John MacGregor, and Steff en Thiel. Configuring software product line features. In ElkePulvermller, Andreas Speck, James Coplien, Maja D Hondt, and Wolfgang De Meuter, editors, *Proceedings of the ECOOP2001 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, Institutf&uuml;rProgrammstrukturen und Datenorganisation, 2001.

[11] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans.Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[12] MikolsJanota and Joseph Kiniry. Reasoning about feature models in higher-order logic. In *SPLC*, pages 13–22. IEEE Computer Society, 2007.

[13] 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, Carnegie-Mellon University Software Engineering Institute, November 1990.

[14] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, January 1998.

[15] Mike Mannion. Using first-order logic for product line model validation. In *Proceedings of the Second International Conference on Software ProductLines*, SPLC 2, pages 176–187, London, UK, 2002. Springer-Verlag.

[16] David L. Parnas. Software fundamentals. chapter On the design and development of program families, pages 193–213. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[17] Shamim Ripon, A unified tabular method for modeling variants of Software product line,SIGSOFT soft. Eng. Notes,37(2),May 2012

[18] J. Rumbaugh, I. Jacobson, and G. Bosch. *The Unified Modeling Language,Reference Manual*. Addison-Wesley, 1999.

[19] Jing Sun, Hongyu Zhang, and Hai Wang. Formal semantics and verification for feature modeling. In *Proceedings of the 10th IEEE International Con-ference on Engineering of Complex Computer Systems*, ICECCS ’05, pages303–312, Washington, DC, USA, 2005. IEEE Computer Society.

[20] Pablo Trinidad, David Benavides, Antonio Ruiz-Cort´es, Sergio Segura, and Alberto Jimenez. Fama framework. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC ’08, pages 359–, Washington,DC, USA, 2008. IEEE Computer Society.

[21] Wei Zhang, Haiyan Zhao, and Hong Mei. A Propositional Logic-Based Method for Verification of Feature Models. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Formal Methods and Software Engineering*, vol-ume 3308 of *LNCS*, chapter 16, pages 115–130. Springer Berlin / Heidelberg, 2004.