UNIVERSITAS INDONESIA

FEATURE GROUPING USING ABSTRACT BEHAVIORAL SPECIFICATION LANGUAGE

PROPOSAL TESIS

REZA MAULIADI
1506782404

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI MAGISTER ILMU KOMPUTER
DEPOK
JUNI 2016

UNIVERSITAS INDONESIA

FEATURE GROUPING USING ABSTRACT BEHAVIORAL SPECIFICATION LANGUAGE

PROPOSAL TESIS
Diajukan sebagai salah satu syarat untuk memperoleh gelar
Magister Ilmu Komputer

REZA MAULIADI
1506782404

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI MAGISTER ILMU KOMPUTER
DEPOK
JUNI 2016

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

CHAPTER 1
INTRODUCTION

This chapter describes about the background of the research, research questions, research goals, benefits of research, scopes of research, and outline of this thesis proposal.

1.1 Background

Software is a very essential need for some people or company in this modern era. It is not only used to support a large processes, but also to support the daily activities of its users. Software contains several features which describe the functions of the software. The need for the features could be vary among users. Even for the same type of software, not all features are really needed by the user. That is a challenge for software developers to create software which fits the needs of each users.

Instead of building software from scratch, it will be great if the developers can build software from reusable components. It can reduce the effort of developers to build a software for a user. One concept which relies on building software from reusable components is Software Product Line (SPL) [**?**].

In Software Product Line (SPL), a software is built from reusable components instead of from scratch so that software development can be done more efficiently [**?**]. The software will be built according to the specific needs of each user by using components that are already prepared before. It can reduce the effort of developers in building software for each user. One language which supports SPL is Abstract Behavioral Specification (ABS).

Abstract Behavioral Specification (ABS) is a modeling language which supports variability by using the concept of SPL [**?** **?** **?**]. ABS uses feature model to express variability and to organize features [**?**]. Feature model is a set of logical constraints that are used to express the dependencies between features. Feature model is represented as a tree of nested features. The example of feature model represented in feature diagram [**?**] shown in Figure 1.1.

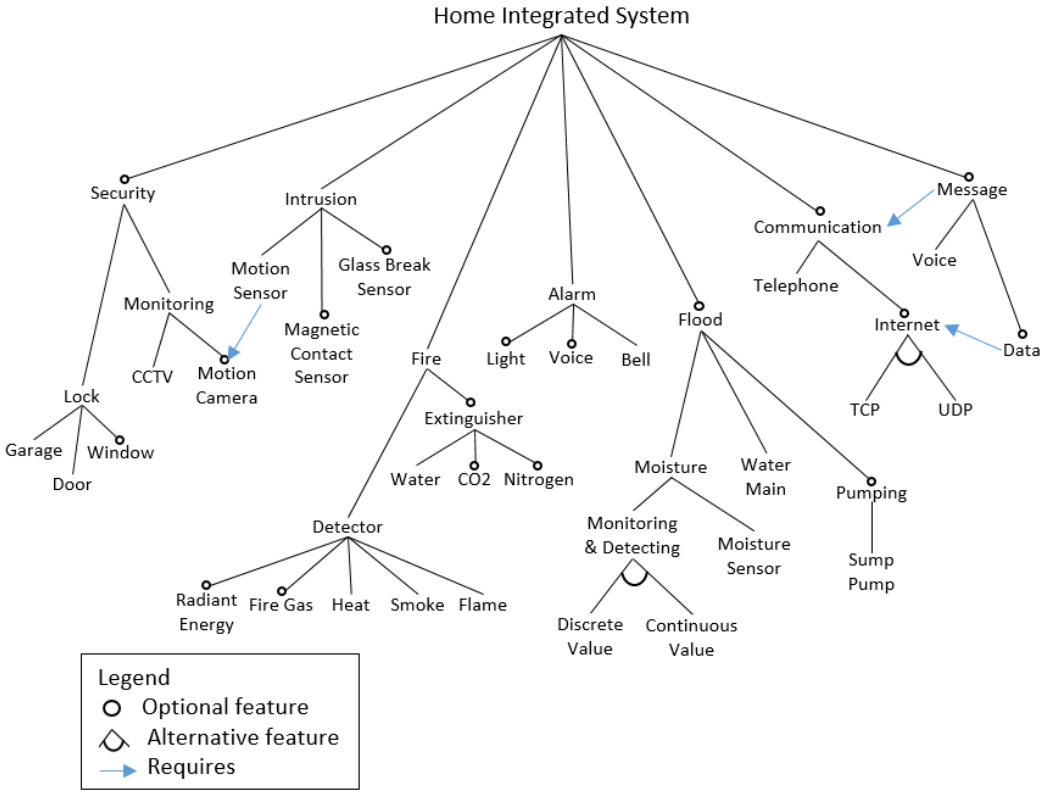

Figure 1.1: Home Integration System Feature Diagram

Source: [**?**] (with additional changes)

The figure above shows a set of features for Home Integrated System (HIS) [**?**]. There are several features which can be chosen by users, such as *Fire, Flood, Alarm, Intrusion*, etc. Some features are general feature which consist of or implemented by the features followed. An *Alarm* feature consists of *Bell, Voice*, or *Light* feature. Then, an *Internet* feature implemented by *TCP* or *UDP* feature.

A feature model is organized based on the visible characteristics of software [**?**]. It can be very large if the number of features available is also increasing. If a feature model has a large number of features, such as Linux kernel with more than 10.000 features [**?**], it can cause the selection of features process be more complex. To address this problem, the features in the feature model can be grouped based on their functions in general. According to [**? ?**], features in the feature model can be grouped into feature binding unit (FBU). FBU is a group of interconnected features based on the relationships that exist in the feature model. By grouping the features, the complexity of the feature model can be reduced. Figure 1.2 shows the example of grouped features represented using feature diagram.

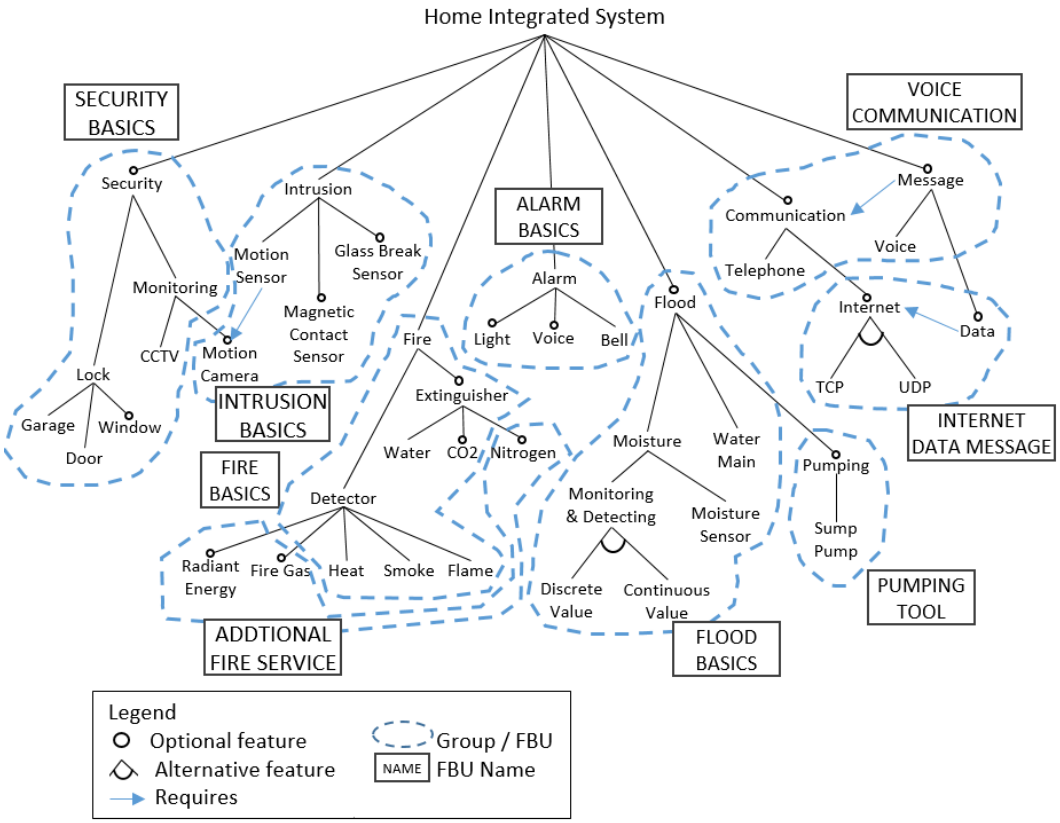

Figure 1.2: Home Integration System Feature Diagram with Grouping
Source: [**?**] (with additional changes)

There are several groups of basic features, as shown on the figure, such as *Security Basics, Intrusion Basics, Fire Basics, Flood Basics,* and *Alarm Basics*. Moreover, there are features grouped to be additional features, such as *Additional Fire Services, Voice Communication, Pumping Tool,* and *Internet Data Message*. The features grouped by the common services they provide and the relationship they have.

Until now, the selection of features to be included in a software product in ABS is done by listing the features chosen using product selection language (PSL) [**?**]. It could make the selection of features difficult to do, even more for a user. Feature grouping can help users in selecting which functions they want to implement. Another issue is about the graphical visualization for the selection of features. It also could make a selection of features be more difficult to do by the user.

In this research the feature grouping mechanism will be made and visualized. The visualization is done by using a simple web application as a tool support. The visualization of features is intended for users so they can make a selection of features with easily. Moreover, the purpose of the feature grouping is to reduce the complexity while doing the features selection. The features that have been selected

by the user will also be able to be directly generated into product selection, thereby reducing the effort to make product selection manually.

1.2 Research Questions

Based on the research purpose, the research questions which will be answered by this research are as follows:

1. How to implement feature grouping from feature model in ABS tools?
2. How to make a visualization of the feature grouping?
3. How to improve the efficiency of building software from the grouped features?

1.3 Research Goals

There are several research questions that have been stated. Based on the background of this research and those research questions, there are several goals which will be the target of this research. The research goals are as follows:

1. The feature grouping mechanism from feature model in the ABS tools is implemented.
2. The visualization of the result of feature grouping mechanism is implemented. Thus, this visualization could be a way to improve the efficiency of building software from the grouped features.

1.4 Benefits of Research

Many researchers are contributing in the development of ABS. This research will contribute in the development of ABS because this research goals are to provide the mechanism to group the features and visualize the result.

In practical term, the grouping of features, hopefully, can reduce the complexity while doing the features selection. The visualization of the features selection also gives ease for users to select the features. Then, as stated in the research goals, the visualization could be a way to improve the efficiency of building software from the grouped features.

1.5 Scope of Research

The aim of this research is to make a grouping mechanism for features, visualize them, and generate the selected features using product selection language. To limit the topics discussed in this research, there are scopes of this research as follows:

1. The focus of the work is to implement grouping mechanism, visualization using web application, and generating the selected features. The grouping mechanism will be implemented in the ABS code generator (compiler back-end). Then, the visualization and the product selection generator will be implemented in the web application.

2. Analyze the grouping mechanism, visualization and product selection generator using a case study. The case study used is ERP sales module features. The features are Odoo ERP sales module features with additional exclusions. Several features which are specific to the vendor will be ignored. It is because the ignored features could be not common features in a sales module of ERP.

1.6 Outline

The outline of this thesis proposal is describe as follows:

- Chapter 1 INTRODUCTION
  Chapter 1 describes the introduction of the thesis proposal that includes the background, research questions, research goals and scope, and the systematic of writing.

- Chapter 2 LITERATURE REVIEW
  Chapter 2 contains the literature review and theories which used in this research as the basis to support. This chapter explains software product lines, Abstract Behavioral Specification (ABS), and feature grouping.

- Chapter 3 PROPOSED GROUPING TECHNIQUE
  Chapter 3 discuss about the grouping technique from the feature model which can be used for selecting the features and generating the selected features.

- Chapter 4 CASE STUDY
  Chapter 4 explains about the case study used in this research which is an enterprise management application sales module features.

CHAPTER 2
LITERATURE REVIEW

This chapter contains the literature reviews which used as the basis of this research. The Software Product Lines, Abstract Behavioral Specification, and feature grouping are explained in this chapter.

2.1 Software Product Lines

Common software development is focused in developing individual software one at a time to individual users (or market segment). The process starts with analyzing the user's needs, then doing the design, implementation, testing, and last deployment of the software. It could take more efforts and times to build software for individual users. To address this challenge, by using the Software Product Lines concept, the development of each software for individual users in similar domain effort and time can be reduce [**?**].

The concept of Software Product Lines (SPL) is to build softwares from reusable software components instead of from scratch so that software development can be done more efficiently [**?**]. By using the components which already prepared before, the software will be built according to the needs of each user. The softwares produced could be similar in one domain, but they are actually different because they are tailored for the specific needs of users [**?**].

There are several promised benefits by using SPL concept [**? ? ?**]. First, the software could be produced faster because the development process which uses reusable components. Second, it could reduce the effort and cost because the reusable components have already done before and the users don't need to pay for the cost of design and development software from scratch. Then, it could improve quality because the reusable components can be checked and tested, either in isolation or in many softwares. Last, the softwares produced could be tailored for specific needs of users.

SPL uses feature-oriented approach in managing variability of the needs or requirements of users. In feature-oriented approach, the variability is represented by using features [**?**]. A feature is a software property that is used to capture commonalities or variabilities among software [**?**]. The variability needs to be modeled in order to manage it. One approach to model the variability is using the feature model

[**?** **?**] and represented graphically by using feature diagram [**?**]. The feature model will be discussed in section 2.1.1 and section 2.1.2.

2.1.1 Feature Model

In SPL, one approach to express variability is by using feature model [**?**]. According to [**?**], a feature model is a model which describes the set of features and the relationships them. By using feature model, a valid choice of features can be shown [**?**]. To represent the feature model in graphical notation, a standard representation is using feature diagram.

2.1.2 Feature Diagram

"A feature diagram is a graphical notation to specify a feature model" [**?**]. A feature diagram is a hierarchical representation like tree representation. In feature diagram, a node consists the feature name denotes a feature. Then, the edges between the node describes their relationships, such as a node connected with empty bullet denotes an optional features and a node connected with filled bullet denotes a mandatory features. If a feature is mandatory, it must be selected whenever the parent feature is selected. Then, multiple a set of child features can be selected only one (alternative) or can be selected more than one (at least 1). The alternative features denoted by empty arch and the 'at least 1' features denoted by filled arch [**?**]. An example of feature diagram is shown in Figure 2.1.

A feature diagram is a tree with the root representing a concept (e.g., a software system) and its descendant nodes being features

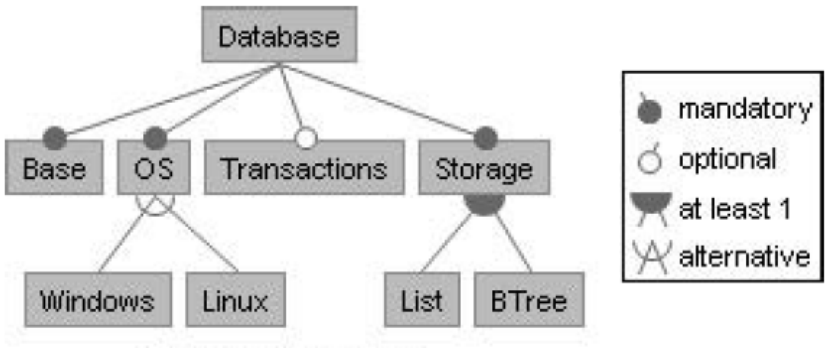

Figure 2.1: Feature Diagram Example of a Small Database Product Line

Source: [**?**]

2.2 Abstract Behavioral Specification

Abstract Behavioral Specification (ABS) is a language which developed for high variability and configurable software such as software product lines (SPL) [**?** **?**]. As said in [**?**], "ABS is designed to fill the niche between design-oriented formalisms such as UML and feature description language FDL, on one hand, and implementation-oriented formalisms such as Spec# and JML, on the other hand". So ABS is a modeling language but it can be executed and generated to other languages such as Java, Maude, or Scala [**?**].

ABS was developed by researchers in HATS (Highly Adaptable and Trustworthy Software using Formal Models) project [**?**] and equipped with ABS tool suite [**?**]. ABS tool suite consists several tools which assist modeling in ABS such as compiler front-end, code generator, ABS plugin, etc. ABS front-end compiler used for code checking and translation code into an internal representation. Then the code generator, use the internal representation and generate it into other languages like Java, Scala, and Maude. To provide editing, visualizing, type checking, and executing, ABS can be integrated with Eclipse IDE which extended by ABS plugin [**?**].

The architecture of ABS is separated into two groups of layers, namely Core ABS and Full ABS [**?**]. Core ABS consists several layers which provides modern programming languages like Java or Scala, concurrency, synchronization, standard contracts, and behavioral interface. Above layers of Core ABS, there are extensions which provide product line engineering, deployment components, and runtime components. These extensions layer called the Full ABS. The architecture of ABS shown in Figure 2.2.

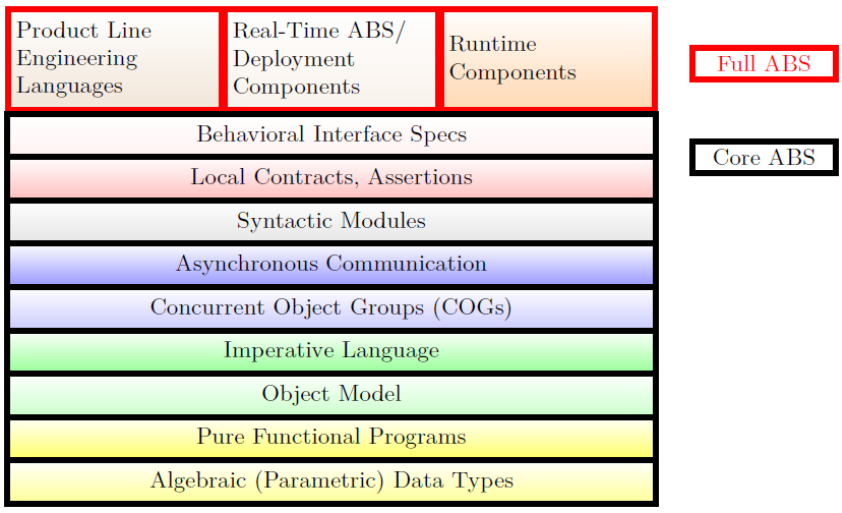

Figure 2.2: The Architecture of ABS Language

Source: [**?**]

In section 2.2.1, a part of Full ABS, which is product line engineering, is explained more. Then a part of ABS tool suite, which is code generator, is explained in section **??**.

2.2.1 Software Product Line in ABS

SPL is an extension from the Core ABS. There are four special languages to represent product line in ABS [**?** **?**], which are:

- Micro Textual Variability Language (μTVL). This language is used to express the variability using feature models. The discussion about feature models in ABS is in section 2.2.1.1.
- Delta Modeling Language (DML). This language is used to express the code-level variability of ABS. Delta used to specify transformation of the core ABS code, such as additions, removals, or modifications some methods or attributes. Delta modeling discussed more in section 2.2.1.2.
- Product Line Configuration Language (CL). This language is used to define the relationships between the feature model and the deltas. Product line configuration discussed more in section 2.2.1.3.
- Product Selection Language (PSL). This language is used to represent the softwares which could be generated. By define the selected features, a software can be generated. Product selection language discussed more in section 2.2.1.4.

The relationship between these languages is shown in Figure 2.3.

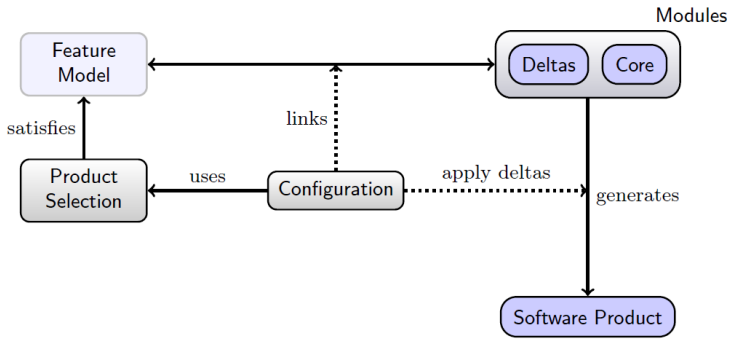

Figure 2.3: Relationship Between the Four Languages

Source: [**?**]

2.2.1.1 Feature Models in ABS

ABS has its feature model. Johnsen [**?**] gives the definition of feature model in ABS below.

> A feature model in ABS is represented textually as a forest of nested features where each tree structures the hierarchical dependencies between related features, and each feature in a tree may have a collection of Boolean or integer attributes. The ABS feature model can also express other cross-tree dependencies, such as mandatory and optional sub-features, and mutually exclusive features.

The feature modeling language in ABS is using micro textual variability language (μTVL). μTVL is an extended subset of textual variability language (TVL). The design of μTVL is simpler than TVL in order to capture the essential feature modeling requirements [**?** **?**]. Beside that, it chosen because μTVL has formal semantics [**?**].

μTVL has a grammar that shown in Figure **??**. For each feature, there is an identifier (which is name), an optional group of sub features, optional attributes, and optional constraints [**?** **?**]. Based on the grammar, FID denotes a name of a feature and then AID denotes a name of an attribute. Each sub feature has a group cardinality:

- **allof**. Indicates that all of the features can be chosen.
- **oneof**. Indicates that just one of the features can be chosen (alternative).
- n_1 .. *. Indicates that the number of features can be chosen is from n_1 to all.
- n_1 .. n_2. Indicates that the number of features can be chosen is from n_1 to n_2.

The *AttributeDelc* clause specifies the declaration of attributes of a feature which can be an integer and boolean. An attribute or a feature could have constraints. That constraints is specified by the *Constraint* clause. There are several restrictions in the *Constraint* clause:

- **ifin:** *constraint;*. Specifies that the *constraint;* will be applied if the current feature **is** selected.
- **ifout:** *constraint;*. Specifies that the *constraint;* will be applied if the current feature **is not** selected.

- **require:** FID;. Specifies that the current feature needs the presence of other feature (which specifies with FID).
- **exclude:** FID;. Specifies that the current feature cannot be chosen if the other feature (which specifies with FID) is chosen, and vice versa.

$$
\begin{aligned}
\mathit{Model} &::= (\texttt{root}\ \mathit{FeatureDecl})^*\ \mathit{FeatureExtension}^* \\
\mathit{FeatureDecl} &::= \mathsf{FID}\ [\{\ [\mathit{Group}]\ \mathit{AttributeDecl}^*\ \mathit{Constraint}^*\ \}] \\
\mathit{FeatureExtension} &::= \texttt{extension}\ \mathsf{FID}\ \{\ \mathit{AttributeDecl}^*\ \mathit{Constraint}^*\} \\
\mathit{Group} &::= \texttt{group}\ \mathit{Cardinality}\ \{\ [\texttt{opt}]\ \mathit{FeatureDecl},\ ([\texttt{opt}]\ \mathit{FeatureDecl})^*\ \} \\
\mathit{Cardinality} &::= \texttt{allof} \mid \texttt{oneof} \mid [n_1\ ..\ *] \mid [n_1\ ..\ n_2] \\
\mathit{AttributeDecl} &::= \texttt{Int}\ \mathsf{AID}\ ; \mid \texttt{Int}\ \mathsf{AID}\ \texttt{in}\ [\ \mathit{Limit}\ ..\ \mathit{Limit}\ ]\ ; \mid \texttt{Bool}\ \mathsf{AID}\ ; \\
\mathit{Limit} &::= n \mid * \\
\mathit{Constraint} &::= \mathit{Expr}\ ; \mid \texttt{ifin:}\ \mathit{Expr}\ ; \mid \texttt{ifout:}\ \mathit{Expr}\ ; \\
&\quad \mid \texttt{require:}\ \mathsf{FID}\ ; \mid \texttt{exclude:}\ \mathsf{FID}\ ; \\
\mathit{Expr} &::= \texttt{True} \mid \texttt{False} \mid n \mid \mathsf{FID} \mid \mathsf{AID} \mid \mathsf{FID.AID} \\
&\quad \mid \mathit{UnOp}\ \mathit{Expr} \mid \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \mid (\ \mathit{Expr}\ ) \\
\mathit{UnOp} &::= ! \mid - \\
\mathit{BinOp} &::= || \mid \&\& \mid -> \mid <-> \mid == \mid != \mid > \mid < \mid >= \mid <= \mid + \mid - \mid * \mid / \mid \%
\end{aligned}
$$

Figure 2.4: Grammar of μTVL
Source: [**?**]

An example of feature model in ABS is shown in Listing 2.1. The example shows the code of feature model for Account features. There are six features, group of sub features, constraints and attributes for the features. It also shows the features that optional (denotes with **opt**) and mandatory (denotes without **opt**).

Listing 2.1: Exampe Account Feature Model in ABS [**?**]

```
root Account {
  group allof {
    Type {
      group oneof {
        Check {ifin: Type.i == 0;},
        Save {ifin: Type.i > 0;
          exclude: Overdraft;}
      }
      Int i;
    },
  opt Fee {Int amount in [0..5];},
  opt Overdraft
  }
}
```

2.2.1.2 Delta Modeling

Delta modeling uses the delta-oriented programming approach. Delta-oriented programming is an approach in SPL as an alternatives to feature-oriented programming approach [**?**]. In delta-oriented programming, the aim is to provide a technique in generating software which modular and flexible in SPL [**?** **?** **?**]. Delta-oriented programming uses *delta modules (deltas)* which the modules are associated with program modifications (add, remove, or modify code) [**?**]. Unlike feature-oriented programming which uses feature modules [**?**] that associated with each features [**?** **?**].

The implication by not associating delta module directly with features, it can obtain the flexibility, better code reuse, and get the ability to resolve conflicts when applying other delta modules [**?**]. Other benefits of deltas is the abilities of delta module. A delta module can add, remove, and modify code.

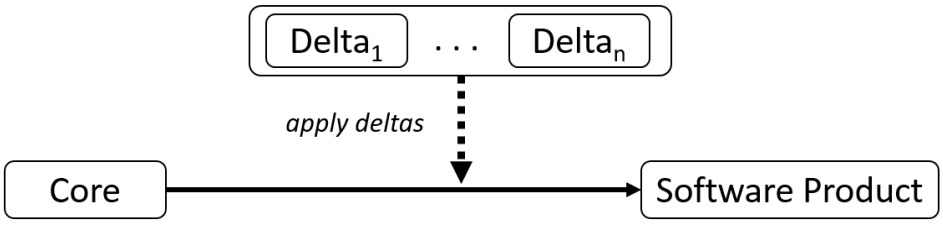

Figure 2.5: The Application of Delta Modules

Source: [**?**]

There are two divisions of implementation of SPL in delta-oriented programming [**?** **?**]. They are core modules and delta modules. The core modules consist classes which implement a complete software for the SPL. Then, to obtain a new variant of software, the delta modules are used to describe how to modify the core modules. The application of delta modules to get a new software variant is shown in Figure 2.5.

2.2.1.3 Product Line Configuration

Product line configuration (CL) uses to link the features in feature model with the delta modules so it can provide a complete specification for the product line [**?** **?** **?**]. The declarations of which deltas applied to which features are written in here. Thus, it can make the process of analyzing the whole product lines more efficient [**?**].

Product line configuration consists set of features and set of delta clause. The connection between feature model, delta module, and the product line configuration

is shown in Figure 2.6. There are three configurations that could be specified in CL [**?** **?**]:

- *application condition*
  The *application condition*, which denoted by **when** clause, associates the delta modules with the features. It describes that if the feature is selected, then use this delta(s) to implement the product line.

- *parameters*
  The *parameters* are derived from the feature attribute values which will be passed to the delta module.

- *order of delta application*
  The *order of delta application*, which denoted by **after** clause, is used to ensure the application of deltas is well-defined and resolve conflicts which could be occurred.

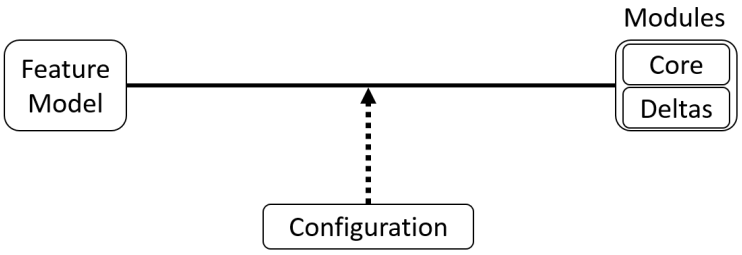

Figure 2.6: The Connection Between Feature Model, Deltas, and CL

Source: [**?**]

An example of product line configuration shown in Listing 2.2. It is a configuration for the Accounts product line (which the features are shown in section 2.2.1.1).

Listing 2.2: Account Product Line Configuration [**?**]

```
productline Accounts;
  features Fee, Overdraft, Check, Save, Type;
  delta DType(Type.i) when Type;
  delta DFee(Fee.amount) when Fee;
  delta DOverdraft after DCheck when Overdraft;
  delta DSave(Type.i) after DType when Save;
  delta DCheck after DType when Check;
```

2.2.1.4 Product Selection Language

The last is to specify the products (softwares) that could be generated using product selection language (PSL). Product selection language describes the features which will be included in the product and gives values to the attributes if needed [**?** **?** **?**]. The syntax of PSL is pretty simple. To specify a product, the elements needed are the product name, the features used in that product, and values of attributes if needed to the features.

The example of product selection language is shown in Listing 2.3. There are examples of products of Accounts product line. For example, *CheckingAccount* product is built from *Type* feature and *Check* feature.

Listing 2.3: PSL of Account Product Line [**?**]

```
product CheckingAccount (Type{i=0},Check);
product AccountWithFee (Type{i=0},Check,Fee{amount=1});
product AccountWithOverdraft (Type{i=0},Check,Overdraft);
product SavingWithOverdraft (Type{i=1},Save,Overdraft);
```

2.2.2 ABS Compiler

ABS has a tool suite that used as a platform for developing and analyzing the ABS code [**?**]. In ABS tool suite, there are two tools that used to be a compiler. They are compiler front-end and compiler back-end. The process to compile the ABS code and generate it to be a other language is shown in Figure 2.7. ABS compiler front-end is used to translate the ABS code into an internal representation, then the ABS compiler back-end will use the internal representation to generate other languages.

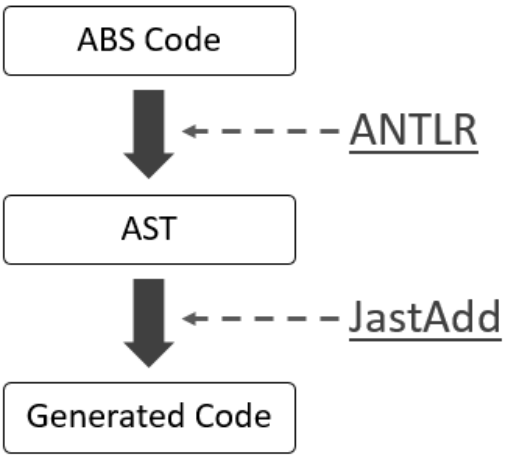

Figure 2.7: ABS Compiler Process Flow

In section 2.2.3, ABS compiler front-end is explained more. Then, the explanation of ABS compiler back-end is in section 2.2.4.

2.2.3 ABS Compiler Front-End

ABS compiler front-end is used to translate the ABS code into an internal representation [**?**]. The internal representation later will be used to check the ABS code for syntax and semantic errors. The compiler front-end takes ABS codes, such as core, feature model, delta modules, configuration, and product selection as inputs and translates them into the internal representation. From the internal representation of feature model, a valid combination of features can be found and used to validate the product selections.

The internal representation is called the Abstract Syntax Tree (AST) [**?** **?**]. AST is a representation of the language constructions and formed as a tree [**?**]. To translate ABS code into AST, the ABS compiler front-end uses a tool called ANTLR. ANTLR is a parser generator which can be used to read, process, execute, or translate structured text or binary files [**?**]. ANTLR also can build the AST by providing grammar annotations. The grammar annotations used to indicate what tokens are to be treated as subtree roots, leaves, or to be ignored with respect to tree construction. To construct a tree with special structure, the ANTLR must be provided with the tree definitions [**?**]. In ABS, it's located in the abstract grammar which specified in `.ast` file.

The abstract grammar is defined in an .ast file called ABS.ast. A partial representation of ABS class hierarchy defined in `ABS.ast` is shown in Figure 2.8.

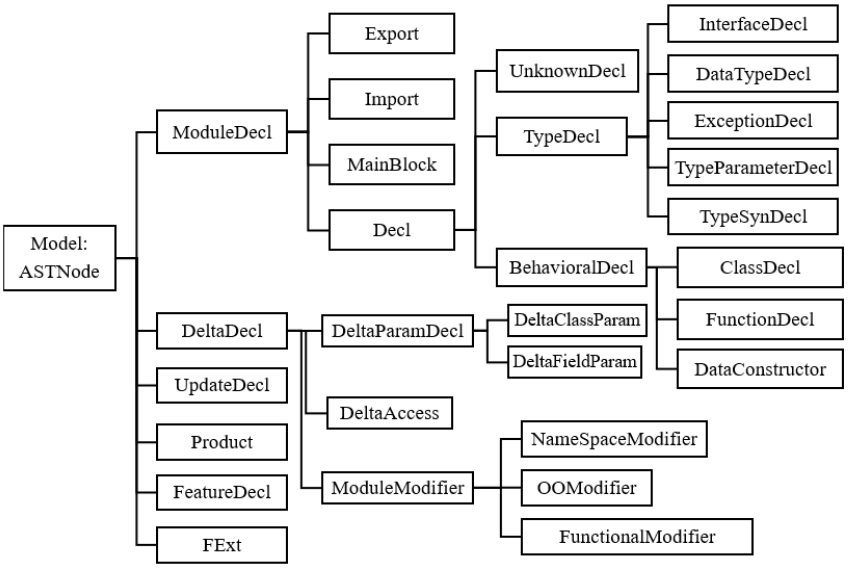

Figure 2.8: ABS.ast Hierarchy Represented in Diagram

2.2.4 ABS Compiler Back-End

The ABS compiler front-end has translated the ABS code into AST, then ABS compiler back-end uses the AST as an input to generate it into other languages. It can generate code to either executable programs in implementation languages, such as Java or Scala, or rewriting systems for simulation and analysis, such as Maude [**?**]. To generate the AST to other languages, ABS uses JastAdd as the tool.

JastAdd is a meta-compilation system which designed to support extensible implementation of compilers and related tools like analyzers or transformation tools [**?**]. By using JastAdd, the compiler can be extended by adding module that contain new abstract syntax and computations on the AST. The abstract syntax is modeled as a class hierarchy. It's modeled from the corresponding code which generated.

JastAdd supports a feature for implementing behavior in the form of attributes [**?**]. Attributes are attached to the AST nodes in AST and the values could be integers, composite values like set, or reference values which are stated using equations. Moreover, the values could point to other nodes in AST or access other attributes [**?**].

The classes in the hierarchy called AST classes because they model nodes in the AST [**?**]. There are method API formed to the classes which correspond to each attributes. By using the API, a programmer can get the correct value of the attribute according to the equation [**?**].

ABS compiler back-end uses the *aspects* of JastAdd to do the code generation. According to [**?**], JastAdd will read the aspects files and weaves the aspect declarations into the appropriate AST classes. By adding a `.jadd` file, the AST classes could be added some ordinary fields and methods [**?**]. Therefore, a behavior could be added to the AST classes such as pretty printing behavior.

2.3 Feature Grouping

A set of features is defined by feature model. Then, the selection of features to be a software product in ABS is done by writing the features included for a software (section 2.2.1.4). The feature model could be consist of feature tree in implementation level. For example, from Figure 1.1 which shown the example of Home Integrated System feature diagram, a *Heat* feature is a child of *Detector* feature which a child of *Fire* feature. The deep level could be vary based on the feature model.

The feature model also can be very large if the number of features available is increasing, such as Linux kernel with more than 10.000 features [**?**]. The selection of features could be more difficult to do. Even more for users to choose the features

they want. To address this problem, the features in the feature model can be grouped based on their functions in general. According to [**?** **?**], features in the feature model can be grouped into feature binding unit (FBU).

2.3.1 Feature Binding Unit

Lee [**?**] define the feature binding unit (FBU) as: "A set of features that are related to each other by the relationships in a feature model". Feature binding consists a set of features that run a common service an must exist together to perform a the correct service [**?** **?**]. As said by Lee [**?**] that the grouping can reduce the complexity in managing the variations, thus it can reduce the complexity for users to choose the common services they want (even if the product selection language still use features to specify a software).

The examples of feature binding units are shown in Figure 2.9. The FBU is a set of features denoted by blue-dotted-line circle. For example, there are several basic features, such as *Security Basics, Intrusion Basics, Fire Basics, Flood Basics,* and *Alarm Basics*. Then there are features grouped to be additional features, such as *Additional Fire Services, Voice Communication, Pumping Tool,* and *Internet Data Message*. The features grouped by the common services they provided and the relationship they had in the feature model.

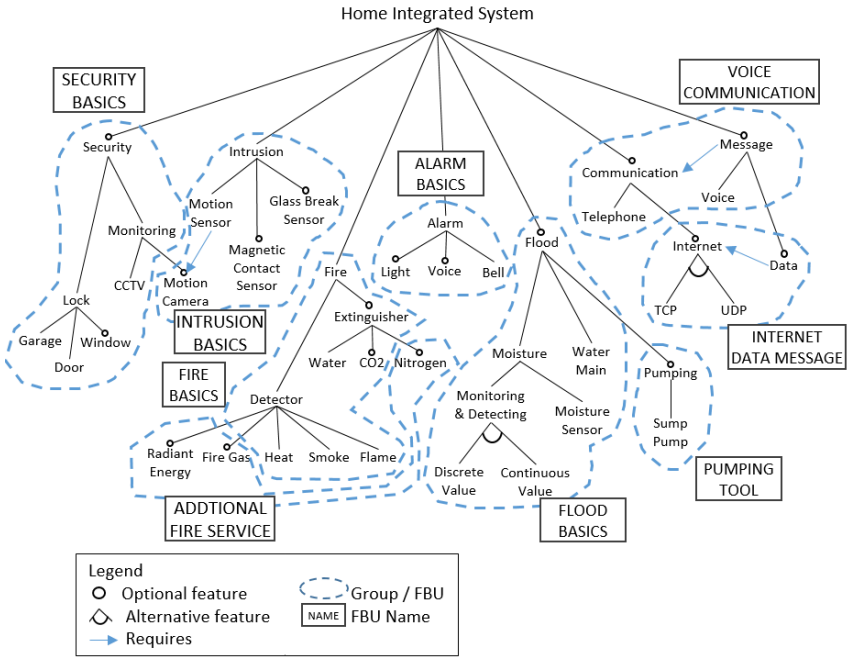

Figure 2.9: Home Integration System Feature Diagram with Grouping

Source: [**?**] (with additional changes)

2.3.2 Feature Grouping Issue

@todo
need more research

CHAPTER 3
PROPOSED GROUPING TECHNIQUE

3.1 Section

3.1.1 Sub Section

3.1.2 Sub Section

CHAPTER 4
CASE STUDY

This chapter discuss about the case study for this research, which is Odoo ERP Sales Modules.

4.1 Odoo Sales Modules

Odoo is an open-source [**?** **?**] enterprise management application [**?**]. It is a suite consists several application modules, such as sales, accounting, manufacturing, purchasing, warehouse management, project management, etc. Odoo is targeting all sizes of companies (small, medium, and large companies). The goals of Odoo are to help the companies to manage, automate, measure and optimize their operations, finances and projects.

As said above, there are several modules in Odoo. One of it is Sales module. In Sales module, there are several features which help the users to do sales activity, such as creating quotation, taking sales order, and managing price-lists. Full Sales module features are explained in Section 4.1.1.

4.1.1 Sales Module Features

Odoo Sales module has 28 features [**?**]. The features are as follow.

- Quotation Builder.
  Build quotation from the predefined products, price lists and templates.
- Quotation template
  Used to design quotation templates that can be reused.
- Upselling
  Quotations are optimized to help users sell more: propose extra options, apply closing triggers, discounts, etc.
- Electronic Signature
  Allow the customers to review and sign their quotations online.
- Contracts
  Record contracts and track invoicing phases, renewal and upselling.

- Sales Orders
  Convert quotation into sales order. Possibility to modify the sales order, remain opened, invoice kits.

- Product Variants
  Create configurable products with multiple variants (size, color, etc.) and options.

- Product Types
  Manage services, physical products to delivery, electronic products and consumables.

- Manage invoicing from sales order
  Invoicing policy is configured in the product but managed from the sales order.

- Price-lists
  Price-list rules used to compute the right price based on customers conditions.

- Dashboard
  Get a full overview on personal activities, next actions and performances.

- Recurring Contracts
  Manage subscriptions with Odoo's recurring contracts: billing, renewal alerts, extra options, MRR dashboard, etc.

- Reduce Data Entry
  Send quotes in just a few clicks, manage pipeline with drag and drop, etc.

- Time and Material Contracts
  Invoice customers based on time and materials with contracts.

- Customer Portal
  Customer can get an access to their quotes to track the status sales orders and delivery orders.

- Milestones
  Manage fixed price contract with invoicing based on milestones. Track invoices, forecast revenues and profitability.

- Discounts
  Apply discounts on every lines.

- Coupons
  Create names coupons or shared one to boost customer demands.
- Order and Invoicing Analysis
  Get statistics based on orders and /or invoices.
- Custom Alerts
  Follow key quotations and orders and get alerts based on relevant activities.
- Email Gateways
  All email communications automatically attached to the right order.
- On Boarding Emails
  Create template of emails for specific product to give information to buyers: access material, reminder of the service, etc.
- Multi company rules
  Automatically mirror sales orders and purchase orders in multi-company setup.
- SaaS KPIs
  Get a dashboard about all SaaS KPIs: Churn, MRR, Lifetime Value, CAC Ratio, upgrades / downgrades, etc.
- Modern User Interface
  A fast user interface designed for sales.
- Mobile
  Sell on the road with Odoo's mobile user interface, working even if the users don't have an internet connection.
- Odoo eSign
  Use Odoo eSign to get signatures on NDAs, contracts or any PDF document.
- Incoterms
  Incoterms appear on the invoice.

4.1.2 Motivation for Odoo Sales Modules Case Study