Journal of

Software Engineering

ISSN 1819-4311

Academic
Journals Inc.

www.academicjournals.com

Journal of Software Engineering 7 (1): 1-29, 2013
ISSN 1819-4311 / DOI: 10.3923/jse.2013.1.29
© 2013 Academic Journals Inc.

An Approach for Validating Feature Models in Software Product Lines

Guoheng Zhang, Huilin Ye and Yuqing Lin
School of Electrical Engineering and Computer Science, University of Newcastle, Callaghan 2308, NSW, Australia

Corresponding Author: Guoheng Zhang, School of Electrical Engineering and Computer Science, University of Newcastle, Callaghan 2308, NSW, Australia

ABSTRACT

In a Software Product Line (SPL), a feature model is widely used to represent the commonalities and variabilities of a family of software products. In the process of establishing feature models, the incorrect and inaccurate feature relationships will lead to feature model errors which prevent the effective product configuration. The feature model validation aims to identify the errors existing in a feature model and find the solutions of resolving the errors. The current validation approaches transformed a feature model into a Constraint Satisfaction Problem (CSP) and used off-the-shelf solvers to reason on the CSP. However, the use of solvers might take an infeasible amount of time for validating large scale feature models, as CSP exhibits the exponential complexity and requires a combination of heuristics and combinational search methods. This study developed an efficient validation approach based on the contradictory feature relationships behind the errors. As the contradictory feature relationships were found based on feature relationship propagation, the solvers were not required by this approach. The performance and correctness of this proposed approach were evaluated by comparing with the CSP based approach based on a set of pre-designed feature models and a number of large-scale feature models.

Key words: Software product line, feature model errors, feature model validation, constraint satisfaction problem, propagation

INTRODUCTION

A software product line is defined as "a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market or mission and that are developed from a common set of core assets in a prescribed way" (Clements and Northrop, 2002). As an important artifact in software product lines, a feature model is widely used to represent the commonalities and variabilities of software product line members and configure specific products in product configuration (Asikainen *et al*., 2006). The feature relationships among features, including hierarchical relationships and cross-tree constraints, are introduced into a feature model to constrain the set of products that can be configured from a feature model. In product configuration, a specific product can be derived from a feature model by selecting the desired features based on the customers' requirements and the selection constraints imposed by the feature relationships (Rabiser *et al*., 2011).

In feature modelling, the designers will inevitably introduce contradictory feature relationships into a feature model by mistake. The contradictory feature relationships are a set of transmissible

feature relationships which will lead to contradictions in product configuration. For examples, there are three relationships among features A, B and C in a feature model: A requires B; B requires C and C excludes A. Obviously these three feature relationships are contradictory, as the inclusion of A will lead to a contradiction (exclusion of A) through these relationships. The contradictory feature relationships will ultimately result in feature model errors, i.e., dead features and false variable features (Von der MaBen and Lichter, 2004). To ensure the effective product configuration, the errors existing in a feature model and their corresponding causes must be identified in feature model validation. As it is an error-prone and time-consuming task for designers to validate large-scale feature models, the automated support has been recognized as one of challenges in the area of feature model validation (Batory *et al*., 2006).

To provide the automated support, a set of approaches transformed a feature model validation problem into a constraint satisfaction problem and used the off-the-shelf solvers to automate the validation (Trinidad *et al*., 2008; Czarnecki and Kim, 2005; Zhang *et al*., 2004). However, these approaches were quite inefficient, especially when explaining feature model errors, as the constraint satisfaction problem is NP-complete and the complexity of these approaches is exponential to the number of units (features and relationships) of the feature model. The solvers might be unable to resolve the CSP of a large-scale feature model in a feasible amount of time. To overcome this limitation, a set of approaches were proposed to reduce the size of the CSP that needs to be resolved by solvers in different ways. For example, Yan *et al*. (2009) eliminated the validation-irrelevant features and feature relationships from a feature model. Segura (2008) used the atomic sets instead of features in validation. Mendonca (2009) developed a propagation algorithm to handle the hierarchical relationships and used the solvers to deal with the cross-tree constraints. Although these approaches could improve the validation efficiency in some domain-specific cases, their complexity was still exponential to the number of units in a feature model as they all use solvers.

This study aimed to propose a validation approach that can work without the solvers. The study of Mendonca (2009) inspired the idea of adapting the structural properties of a feature model for validation. Based on the observation that feature model errors were caused by the contradictory feature relationships, this study proposed an approach of identifying and explaining feature model errors based on the contradictory feature relationships existing in a feature model. An algorithm of finding all contradictory feature relationships from a feature model was developed based on feature relationship propagation in a feature tree. Furthermore, this study developed a tool suite called Feature Model Validation Tool (FMVTool) to implement the concepts of the proposed approach and evaluated the proposed approach by comparing the validation results from FMVTool and a CSP based validation tool called Feature Model Analyzer (FAMA) (Benavides *et al*., 2007). The evaluation results showed that the approach proposed in this study can provide the same validation results with the CSP based approach and this proposed approach was more efficient than the CSP based approach for validating large-scale feature models.

BACKGROUND

Feature models: A feature model is a formalism of capturing and representing the common and variable characteristics among the products in a software product line in terms of features (Asikainen *et al*., 2006). Mostly, a feature model is represented by an and/or tree-structure feature diagram which involves of two kinds of components: features and feature relationships. The features describe distinguishable characteristics of software systems and feature relationships represent the selection constraint among features. The feature relationships can be further

classified into hierarchical relationships which describe the selection constraint between a parent feature and its child features and cross-tree constraints which represent the selection constraint between two cross-tree features. Since Kang *et al.* (1990) first introduced the notation of basic feature diagram in Feature Oriented Domain Analysis (FODA); several extensions have been proposed to improve its succinctness and naturalness. For example, Riebisch *et al.* (2002) introduced the Unified Modelling Language (UML) multiplicity to enhance the feature diagram notation. Czarnecki *et al.* (2004) proposed Cardinality-based Feature Model (CBFM) which includes feature cardinalities, group cardinalities and feature diagram references. Benavides *et al.* (2005) extended feature models with feature attributes to represent quality attributes. Among these extensions, all authors agreed that a minimum feature diagram notation should be able to present three kinds of hierarchical relationships: mandatory, optional and feature groups and two kinds of cross-tree constraints: requires and excludes (Mendonca *et al.*, 2008). Therefore, the cardinality-based feature model was most widely adapted in software product line community. This study focused on validating the cardinality-based feature models. The semantics and notations of feature relationships in a cardinality-based feature model are shown in Table 1.

Figure 1 shows an example of the cardinality based feature model from tourist guide software product line. "Tourist guide" is the root feature. If "service" is selected, "route search" must be selected because of the mandatory relationship between them and "position detection" may or may not be selected because of the optional relationship between them. If "terminal device" is selected, at least one from "Mobile" and "PDA" must be selected due to the group cardinality "1... 2".

To maintain the information of the cross-tree constraints, this study adapted a matrix-based approach proposed by Ye (2005). First, a matrix with n columns and n rows is generated

Table 1: Semantics and notations of feature relationships in cardinality based feature models

Feature relationships	Semantic	Notation
Hierarchical relationships		
Mandatory	If the parent feature is selected, the child feature which has a mandatory relationship with its parent must be selected.	Parent, Child
Optional	If the parent feature is selected, the child feature which has an optional relationship with its parent may or may not be selected.	Parent, Child
Variation point and variants	If a variation point (VP) is selected, at least a variant features must be selected and at most b variant features can be selected from the group of variant features based on the cardinality [a... b].	VP, (a-b), Var, Var, ... Var
Cross tree constraints		
Requires	"A requires B" means that if feature A is selected, feature B must be selected.	A, Requires, B
Excludes	"A excludes B" means that A and B cannot be selected into the same member product constraints	A, Excludes, B

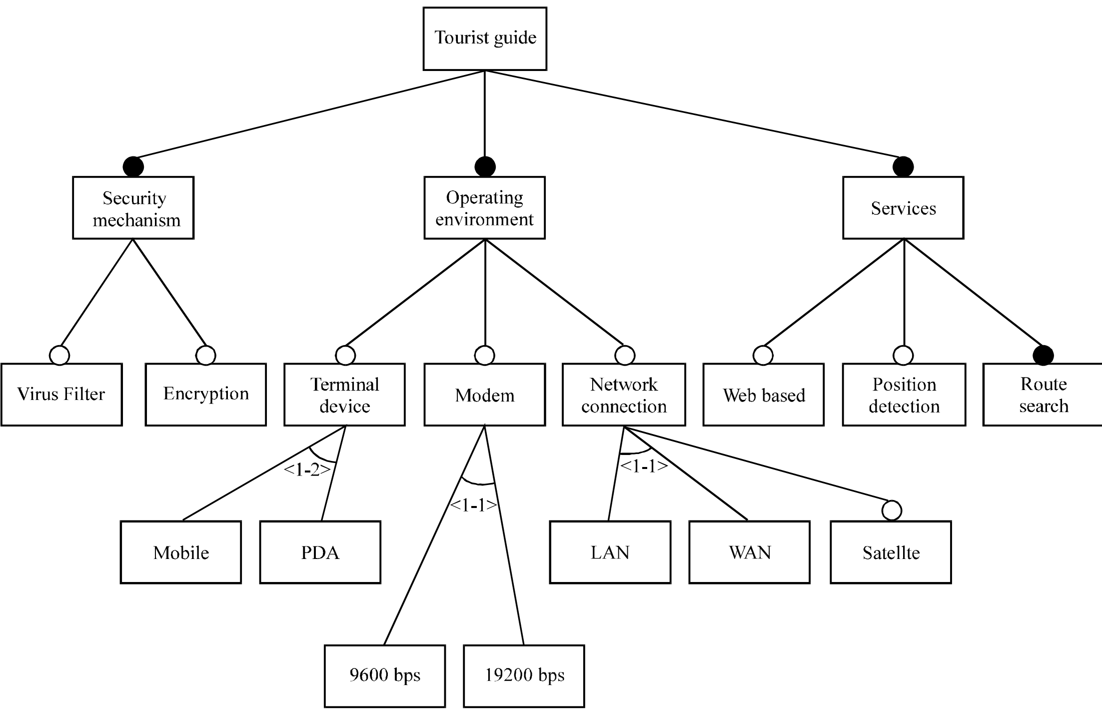

Fig. 1: A feature model of tourist guide software product line

Table 2: Matrix of dependencies in the feature model of tourist guide software product line

	Mobile	WAN	LAN	Modem	Web-based	Terminal device	Encryption
Mobile							2
WAN					1	1	
LAN				1			
Modem						2	
Web-based		1					
Terminal device				2			1
Encryption	2						

where n is the total number of features involved in the cross-tree constraints. Second, all the features that involve in the cross-tree constraints are listed as the column elements and row elements respectively in the generated matrix. Finally, if a feature in the row "requires" a feature in the column, the corresponding cell is assigned with "1", while if a feature in the row "excludes" a feature in the column, the corresponding cell is assigned with "2". Following the above steps, a matrix is designed as shown in Table 2 for managing the cross-tree constraints of the feature model in Fig. 1. This matrix shows that "mobile" excludes "encryption", "terminal device" excludes "modem", "terminal device" requires "encryption" and etc.

The features of a feature model can be classified into full-mandatory features and variable features based on the hierarchical relationships. A full-mandatory feature is a feature which connects with root feature by a hierarchical path where all hierarchical relationships are mandatory (Von der MaBen and Lichter, 2004). Based on the semantics of mandatory relationship, the full-mandatory features will appear in all products in a software product line. The features except the full-mandatory features in a feature model are called variable features in this study. In

the feature model of Fig. 1, the full-mandatory features include Tourist Guide, Security Mechanism, Operating Environment, Network Connection, Services and Route Search and all other features are variable features. As discussed (Ye *et al.*, 2010), the "requires" or "excludes" constraints can only exist between two variable features. The involvement of full-mandatory features in the cross-tree constraints will lead to feature model errors.

The product configuration based on a feature model is the process of selecting the desired features from a feature model based on the customers' requirements. As features are units used for selection, the product configuration based on a feature model is also named as Feature Based Product Configuration (FBPC) (Mendonca *et al.*, 2008). In FBPC, the variabilities in a feature model are provided to the stakeholders who need to understand and resolve them (Rabiser, 2009). Meanwhile, the selection must satisfy the feature relationships specified in the feature model. However, a feature model may include the contradictory feature relationships that will result in contradictions in FBPC. For instance, feature mobile in Fig. 1 is a variable feature which can be provided to customers for selection in product configuration. If mobile is selected based on the customers' requirements, terminal device must also be selected due to the parental relationship between them. In order to satisfy the constraint "terminal device requires encryption", encryption must be selected due to the inclusion of terminal device. If encryption is selected, mobile must be removed to satisfy the constraint "encryption excludes mobile", which is obviously a contradiction since feature mobile have already been selected. Then there is an error related with feature mobile, as its selection will lead to contradictions in FBPC. To ensure effective product configuration, the feature model errors and their causes must be identified in feature model validation.

Feature model validation: Feature model validation aims to identify the errors existing in a feature model and provide explanations for each identified error. Trinidad *et al.* (2008) considered a feature model error as an incorrect definition of feature relationships which lead to that the set of products described by a feature model may not match the software product line it describes. In literature three critical feature model errors have been proposed: void feature model (Benavides *et al.*, 2005), dead feature (Trinidad *et al.*, 2008), false variable feature (Trinidad *et al.*, 2008). Benavides *et al.* (2010) has summarized the definitions of these three feature model errors as follows:

- **Dead features:** A feature is dead if it cannot appear in any product despite of being defined in a feature model
- **False variable feature:** A feature is false variable if it must be present in a product whenever its parent is present, despite of not being modeled as a mandatory feature
- **Void feature model:** A feature model is invalid or void when no products can be derived from the feature model. A void feature model is a special case of dead feature error where all the features are dead

The feature model of Fig. 2 is used to illustrate the above feature model errors. In Fig. 2, R is the root feature and it will be included in all products; feature A, B, C and D connects with R by hierarchical relationships Br-1, Br-2, Br-3 and Br-4 respectively; feature E and F connects with A by optional relationship Br-5 and Br-6; feature G and H constitute a feature group with cardinality "1..1" and connect with D by Br-7; feature E excludes with feature B by the "excludes" constraint

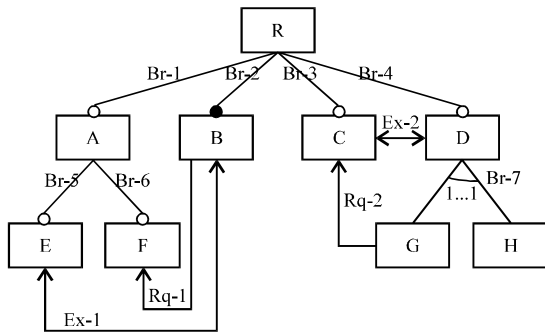

Fig. 2: An example of dead features and false variable features in a feature model

"Ex-1"; feature C excludes with feature D by the "excludes" constraint "Ex-2"; feature B requires feature F by "requires" constraint "Rq-1"; and feature G requires feature C by "requires" constraint "Rq-2".

Based on the definitions of feature model errors, a set of errors in the feature model of Fig. 2 can be detected. Feature G and E are dead features because they cannot appear in any product. Feature H and F are false variable features because they must be present in all products. The causes of these errors can be found by examining the feature relationships. As discussed (Ye *et al.*, 2010), the full-mandatory features cannot be involved in the "requires" or "excludes" constraints, as they are not selectable. Therefore, feature E is dead because it excludes with the full-mandatory feature B and feature F is false variable because it is required by the full-mandatory feature B. The errors on G and H lie in the contradictory feature relationships. For example, feature G is dead due to the set of contradictory feature relationships: Rq-2, Ex-2 and Br-7; feature H is false variable due to the set of contradictory feature relationships: Br-7, Rq-2 and Ex-2. The feature model of Fig. 2 is not a void feature model because at least one product which consists of R, B and F can be derived from the feature model.

As discussed above, there are two important tasks that need to be accomplished in feature model validation: error identification and error explanation:

- **Error identification:** In the task of error identification, a feature model is input and a list of feature model errors are returned, i.e., dead features, false variable features and void feature model. For instance, if the feature model of Fig. 2 is input for error identification, four errors will be returned: dead feature G, dead feature E, false variable feature H and false variable feature F
- **Error explanation:** In the task of error explanation, a feature model error is input and a set of its explanations are returned. Herein, an explanation of a feature model error is a right set of feature relationships that must be modified to remove the error (Trinidad *et al.*, 2008). Each feature model error may correspond to a set of explanations. For instance, if the error "dead feature G" is input for error explanation, three explanations will be returned: {Br-2}, {Ex-2} or {Br-7}

RELATED WORKS

The feature model validation was first identified in FODA (Kang *et al.*, 1990) as a fundamental task for feature modelling in software product lines. Since then a set of validation approaches have

been proposed. For example, Zhang *et al.* (2004) transformed a feature model into propositional formulas and used Support Vector Machine (SVM) system to detect conditional dead features and conditional full-mandatory features. Czarnecki and Kim (2005) transformed a feature model into a Constraint Satisfaction Problem (CSP) and used CSP solvers to identify void feature model and dead features. Trinidad *et al.* (2008) also transformed a feature model into a constraint satisfaction problem and used CSP solvers to detect the explanations of a feature model error.

The above approaches formalize a validation problem into a Constraint Satisfaction Problem (CSP) or Boolean satisfaction problem (SAT) which is a certain form of CSP where the domains of variables are Booleans. Then the off-the-shelf tools (i.e., SAT-solvers, CSP-solvers and Binary Decision Diagram (BDD)-solvers) are used to automate the feature model validation. However, the CSP is NP-complete in theory (Cook 1971) and it exhibits exponential complexity (Yan *et al.*, 2009; Segura, 2008; Mackworth, 1977), requiring a combination of heuristics and combinational search methods. Due to the NP-complete nature, the validation of a large scale feature model in the above approaches suffers from state space explosion problem, especially when explaining feature model errors. For example, Trinidad *et al.* (2008) transformed a feature model into a constraint satisfaction problem where all features are transformed into feature variables and all feature relationships are transformed into constraint variables. Then the feature model errors and their explanations were identified based on whether valid solutions can be derived from the constraint satisfaction problem by solvers under the certain conditions. Using Trinidad's approach, to validate a feature model with n features and m feature relationships, the searched space of the constraint satisfaction problem is as large as 2^{m+n} (Trinidad *et al.*, 2008). In this case, it is impossible to explore the entire state space with limited resources of time and memory by the CSP solvers.

To improve the efficiency, a set of approaches aimed to reduce the size of the CSP that needs to be resolved by the solvers without changing the result of feature model validation. For example, Yan *et al.* (2009) proposed an approach of eliminating the validation-irrelevant features and feature relationships from a feature model. Segura (2008) proposed an approach of finding atomic sets, the group of features that can be treated as a unit in product configuration and using atomic sets instead of features in feature model validation. Mendonca (2009) developed a propagation algorithm to deal with the hierarchical relationships and used the SAT-solvers to reason on the extra-constraints, such as requires and excludes. As discussed earlier, the complexity of resolving the CSP of a feature model by solvers is exponential to the number of units (features and relationships) of the feature model. In this sense, these approaches can improve the validation efficiency in some domain-specific cases. For example, the validation efficiency can be improved if the number of validation-irrelevant units is large in Yan's study.

In conclusion, the current validation approaches were inefficient, as they need to use the solvers to explore the entire solution space of a constraint satisfaction problem. Although a set of approaches improved the validation efficiency by reducing the size of the CSP that needs to be resolved by solvers without changing the validation results in different ways, the complexity was still exponential to the number of model elements due to the use of solvers. The limitations of the current approaches inspired us to develop a more efficient validation approach that can work without the solvers.

AN APPROACH FOR VALIDATING FEATURE MODELS

It is argued that a feature diagram has important structural properties that will support efficient algorithms for validating a feature model. The main idea is based on the observation that

feature model errors are caused by the contradictory feature relationships. Based on this observation, a feature model validation problem can be transformed into a problem of finding Contradictory Feature Relationships (CFR) from a feature model. Then the feature model errors and their corresponding explanations can be identified based on the found CFR. A recursive algorithm of finding CFR from a feature model is developed based on the feature relationship propagation on feature trees.

Cause of feature model errors: This section aims to find the causes behind feature model errors. As discussed by Ye *et al.* (2010), the full-mandatory features involved in the cross-tree constraints will result in feature model errors. For example, if a variable feature excludes with a full-mandatory feature, it will be a dead feature as it cannot appear in any product, such as feature E in Fig. 2. If a variable feature is required by a full-mandatory feature, it will be a false variable feature as it will appear in all products, such as feature F in Fig. 2. The feature model errors in this case can be resolved in two ways: removing the cross-tree constraints imposed on full-mandatory features or modifying the hierarchical relationships to change full-mandatory features into variable features. Once all feature model errors caused by the cross-tree constraints imposed on full-mandatory features are resolved, the feature model is not void, because at least one product which consists of all the full-mandatory features can be derived from the feature model.

Besides the errors in the above case, it is observed that feature model errors (dead feature or false variable feature) are caused by contradictory feature relationships. The “dead feature” is used as an example to illustrate this idea. A dead feature cannot be present in any product, which means it cannot be selected in product configuration. This is because the selection of a dead feature will result in contradictory configuration conclusions through feature relationship propagation. Herein, the feature relationship propagation starting from a specific feature f is the process of propagating the inclusion or removal of f to a set of other features by the transmissible feature relationships. A contradictory configuration conclusion is a conflict in feature relationship propagation and arises in two scenarios: a feature that has already been included needs to be removed or a feature that has already been removed needs to be included. For a specific feature f, if its inclusion or removal results in contradictory configuration conclusions through a set of feature relationships, there is an error on feature f and this set of feature relationships is one cause of the error. This study focuses on resolving the feature model errors caused by contradictory feature relationships and define dead features and false variable features based on their causes as follows:

- **Dead feature:** A feature f is a dead feature if the inclusion of f will result in contradictory configuration conclusions by feature relationship propagation starting from f
- **False variable feature:** A feature f is a false variable feature if the removal of f will result in contradictory configuration conclusions by feature relationship propagation starting from f

Based on the above discussion, the key issue of identifying an error on feature f is to find the contradictory configuration conclusions through the feature relationship propagation starting from f. The key issue of explaining an error on f is to find the cause of the error, which is the set of feature relationships that will result in the contradictory configuration conclusions. In this study, a set of feature relationships that will result in a contradictory configuration conclusion is called a Contradictory Relationship Set (CRS). CRS (f, inclusion) is used to represent a CRS by which the inclusion of f can result in a contradictory configuration conclusion and CRS (f, removal) is used

to represent a CRS by which the removal of f can result in a contradictory configuration conclusion. In some cases, the inclusion or removal of a specific feature f will result in more than one contradictory configuration conclusion. As each contradictory configuration conclusion is caused by a specific CRS, the corresponding error on f is caused by all its related CRS. For example, in Figure 2, the feature G is a dead feature and its CRS is CRS (G, inclusion) = {Rq-2, Ex-2, Br-7}.

Based on the causes of feature model errors, the formal rules of identifying feature model errors are proposed as follows:

- If there exists one or more CRS (f, inclusion) in a feature model, feature f is a dead feature
- If there exists one or more CRS (f, removal) in a feature model, feature f is a false variable feature

In order to identify feature model errors and find their explanations, all Contradictory Relationship Sets (CRS) existing in a feature model need to be found. As a CRS is found whenever a contradictory configuration conclusion arises in feature relationship propagation, the process of feature relationship propagation and the detailed propagation rules are introduced in next section.

Feature relationship propagation: The feature relationship propagation is the process of propagating the inclusion or removal of a specific feature to other features in the feature model through different feature relationships, such as hierarchical relationships and cross-tree constraints. In feature relationship propagation starting from a specific feature f, the inclusion or removal of f will lead to that its adjacent features, such as parent, child and dependent features, are included or removed based on their specific relationship with f. Then the inclusion or exclusion of its adjacent features will result in inclusions or exclusions of their adjacent features as well and so forth. Then the key issue of the feature relationship propagation is the rules about how to propagate the inclusion or removal of a specific feature to its adjacent features. To propose propagation rules, some definitions are first given:

Definition 1: Each feature in a feature model has an attribute named as "status" that ranges over the domain {-1, 0, 1}. The default "status" value of a feature is "0" which means a decision has not been made on this feature. In product configuration, the attribute "status" gets the value "1" if this feature is included to be a part of the product and gets the value "-1" if this feature is removed from the product.

Definition 2: In product configuration, the action of changing the "status" value of a feature f from "0" to "1" is named as including f and represent as "Include (f)". The action of changing the "status" value of a feature f from "0" to "-1" is named as removing f and represented as "Remove (f)".

Definition 3: In a feature model, a feature group fg can be represented as $\{f_1, f_2 ... f_n\}$. The number of included features in fg is represented as Number of Included (fg). The number of removed features in fg is represented as Number of Removed (fg). The total number of features in fg is represented as Number of Features (fg).

Definition 4: The adjacent features of a feature f includes all the features that connect with f by a single feature relationship, such as mandatory, optional, feature group cardinality, requires or excludes. The adjacent features of feature f are represented as follows:

- **Parent feature:** f has a parent feature if f is not root feature of a feature model. f. parent is used to represent the parent feature of f. If f is a root feature, f. parent has the value "null". f. pr is used to represent the hierarchical relationship that connects feature f with its parent feature:

  - f. pr = m, if a mandatory relationship connects f with f. parent
  - f. pr = o, if an optional relationship connects f with f. parent
  - f. pr = v, if a variation point relationship connects f with f. parent

- **Child feature:** f has a number of child features if f is not leaf feature of a feature model. f.child is used to represent the set of child features of feature f. If f is not a variation point feature, f.child includes two subsets f.child-mandatory and f. child-optional while if f is a variation point feature, f.child includes one subset f. child-variants:

  - The notation "f. child-mandatory" represents the set of child features which connect with f by mandatory relationships
  - The notation "f. child-optional" represents the set of child features which connect with f by optional relationships
  - The notation "f. child-variants" represent the set of variant features that connect with f by variation point cardinality

- **Brother feature:** f has a set of brother features if f is a variant feature under a variation point. f. brother is used to represent the set of variant features which belong to the same variation point with f
- **Friend feature:** f has a set of friend features if f connects with other features by "requires" or "excludes" feature dependencies. f. friend is used to represent the set of features which connect with f by cross-tree feature dependencies. The feature set f. friend can be decomposed into three subsets f. exld, f. reqed and f. reqs where f. friend = f. exld∪f. reqed∪f. reqs:

  - The notation f. exld represents the set of features that f excludes
  - The notation f. reqed represents the set of features that require f
  - The notation f. reqs represents the set of features that f requires

Based on the above definitions, the adjacent features for any specific feature in a feature model can be identified. The tourist guide feature model of Fig. 1 is used to show how to represent the adjacent features of a specific feature in a feature model. The adjacent features of two features "Terminal Device" and "LAN" are given in Table 3. In Table 3, {∅} means the feature in the column has no adjacent features of the type in the row.

In order to simplify the feature model, the optional relationships are first transformed into variation point relationship. In cardinality-based feature model, a feature f and its optional child features f. child-optional can be transformed into a variation point with cardinality [0...n] where

Table 3: Example of adjacent features in tourist guide software product line

Feature f	Terminal device	LAN
f. parent	{Operating environment}	{Network connection}
f. child-mandatory	{ø}	{ø}
f. child-optional	{ø}	{ø}
f. child-variants	{Mobile, PDA}	{ø}
f. brother	{ø}	{WAN}
f. exld	{Modem}	{ø}
f. reqed	{WAN}	{Web-based}
f. reqs	{Encryption}	{Terminal device, web-based}

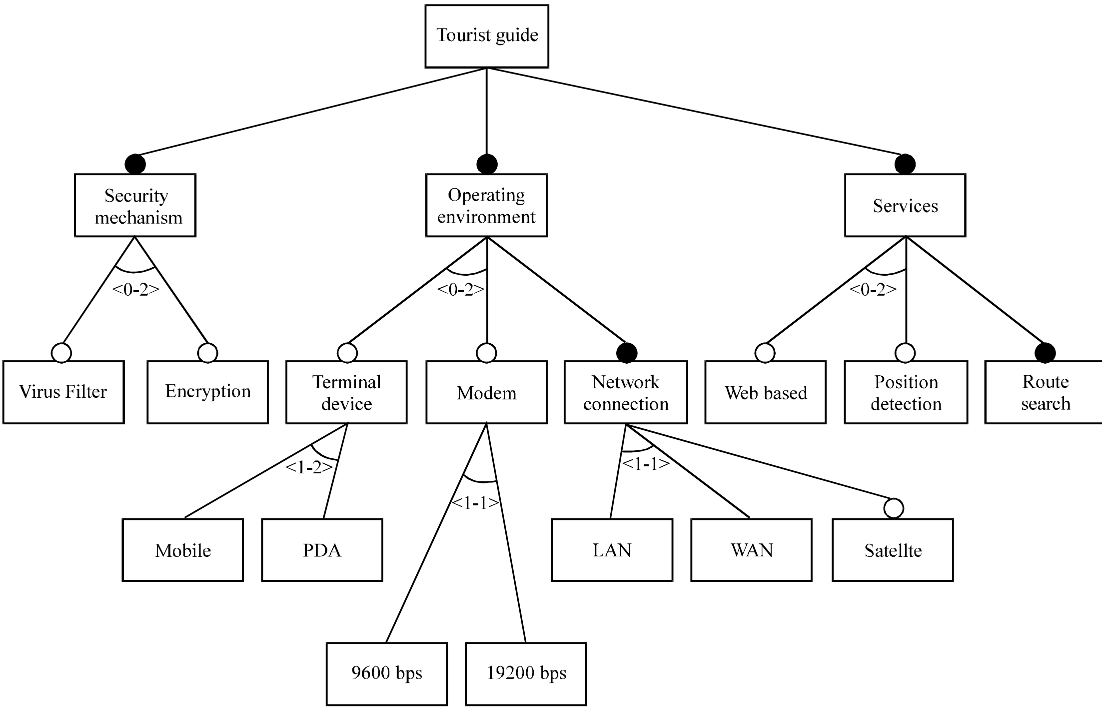

Fig. 3: Transferred feature model of tourist guide software product line

n is the number of optional child features. For instance, Fig. 3 shows the tourist guide feature model after transforming all optional features into variation points. The virusfilter and encryption are two optional features under security mechanism in Fig. 1 and they are transformed into a variation point with cardinality [0... 2] in Fig. 3. The satellite is an optional feature under network connection in Fig. 1 and it is transferred into a variation point with cardinality [0... 1] in Fig. 3. The feature model of Fig. 1 is same with feature model of Fig. 3, as they indicate the same set of products. After the above transformation, only four kinds of feature relationships need to be considered in feature relationship propagation process: requires, excludes, mandatory and variation point with cardinality.

In feature relationship propagation, the inclusion or removal of a specific feature has different impacts on its adjacent features based on the semantics of different feature relationships specified in Table 1. In the following, the detailed rules of feature relationship propagation are defined. In these propagation rules, "Include (f)" is true if f is included into a product while "Remove (f)" is true if f is removed from a product. The propositional logic "A⇒B" means if A is true, B will be true.

1. When the action imposed on f is "include (f)" which means f is included into a product, the adjacent features of f will be affected through feature relationships as follows:

- **Parent feature:** When f is not the root feature, if f is included, its parent feature will be included. The propagation rule is:

Include (f)∧(f. parent != null)⇒Include (f. parent)

- **Mandatory child feature:** When f is not leaf feature and f has a set of mandatory child features f.child-mandatory, if f is included, all features in f.child-mandatory will be included. It should be noted that a variation point feature is the mandatory child feature of its parent feature

Include (f)⇒∀fcm∈f.child-mandatory: Include (fcm)

- **Variant child feature:** If f is a variation point feature and it has a group of variant child features f.child-variants with group cardinality of [a...b], when f is included and the number of removed features in f.child-variants reaches the maximum allowed value (NumberofFeatures (f.child-variants)-a), all features that are not determined in f.child-variants will be included. The propagation rule is:

Include (f) ∧number of features (f.child-variants) -a = = NumberofRemoved (f.child-variants) ⇒∀fcv∈f.child-variants ∧fcv. status = = 0:Include (fcv)

- **Brother feature:** When f is a variant feature under a variation point feature with cardinality a...b and it has a set of brother features f.brother, if f is included and the number of included features in f.brother reaches its maximum value (b-1), all the features in f.brother that are not determined will be removed. The propagation rule is:

Include (f) ∧NumberofIncluded (f.brother) = = b-1⇒∀fb∈f.brother ∧fb. status = = 0:Remove (fb)

- **Requires friend feature:** When there exists a set of features that f requires, if f is included, all features in f. reqs will be included. The propagation rule is:

Include (f)⇒∀fr∈f.reqs:Include (fr)

- **Excludes friend feature:** When there exists a set of features that f excludes, if f is included and all the features in f. exld will be removed. The propagation rule is:

Include (f)⇒∀fr∈f.exld:Remove (fr)

2. When the action on f is Remove (f) which means f is removed from products, the adjacent features of f will be affected through feature relationships as follows:

- **Mandatory parent feature:** When f is not the root feature and f connects with its parent feature by mandatory relationship, if f is removed, its parent feature f. parent will be removed. The propagation rule is:

Remove (f) $\wedge$f. pr = = m$\Rightarrow$Remove (f. parent)

- **Variation point parent feature:** When f is a variant feature under a variation point feature with cardinality [a... b] and it has a set of brother features f. brother, if f is removed and the number of removed features in f.brother exceeds its maximum value NumberofFeatures (f.brother)-a, its parent feature f. parent will be removed. The propagation rules is:

Remove (f) $\wedge$NumberofRemoved (f.brother)>NumberofFeatures (f.brother) -a$\Rightarrow$Remove (f.parent)

- **Child feature:** When f is not a leaf feature and it has a set of child features f. child, if f is removed, all the features in f. child will be removed. The propagation rule is:

Remove (f)$\Rightarrow\forall$fc$\in$f.child:Remove (fc)

- **Brother feature:** When f is a variant feature under a variation point with cardinality a... b and it has a set of brother features f.brother, if its parent feature f. parent has already been included and the number of removed features in f.brother reaches its maximum value NumberofFeatures (f.brother)-a, all the features that are not determined in f.brother will be included. The propagation rule is:

Remove (f) $\wedge$NumberofRemoved (f.brother) = = NumberofFeatures (f.brother) -a $\wedge$f. parent. status = =1$\Rightarrow\forall$fb$\in$f.brother $\wedge$fb. status = = 0:Include (fb)

- **Friend feature:** When there exists a set of features that requires f, if f is removed, all the features in f.reqed will be removed. The propagation rule is:

Remove (f)$\Rightarrow\forall$fr$\in$f.reqed:Remove (fr)

Based on the above propagation rules, the feature relationship propagation starting from a specific feature f can be achieved in an iterative manner. At the first propagation step, the inclusion or removal of a specific feature f will lead to that its adjacent features, such as parent, child and dependent features, are included or removed. Then at each propagation step, the features that are included or removed at the previous step will propagate their inclusions or removals to their adjacent features. Finally, the feature relationship propagation terminates if no features are included or removed in one propagation step.

One example is used to illustrate the process of feature relationship propagation. In the feature model of Fig. 4, the inclusion of feature f will propagate to a set of its adjacent features: E is removed due to the relationship SR-1; C is included due to the relationship RQ-3; and A is included due to the relationship SR-1. Then at step two, the features included or removed at step one, including E, C and A, will propagate to their adjacent features as well: the removal of E results in the removal of D because of RQ-2; and the inclusion of A results in the inclusion of B and R because of RQ-1 and BR-1, respectively. At step three, the newly included or removed features D, B and R will propagate to their adjacent features, but only the inclusion of R results in the inclusion of K

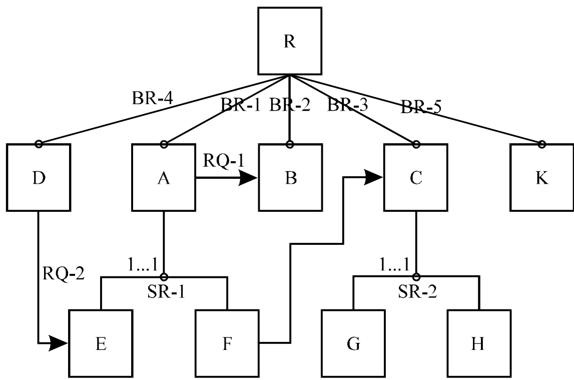

Fig. 4: A Feature model for illustrating feature relationship propagation

by BR-5 at this step. As no features will be removed or included because of the inclusion of K at step four, the feature relationship propagation terminates. In conclusion, the feature relationship propagation starting from the inclusion of f in the feature model of Fig. 4 will lead to the inclusion of C, A, B, R, K and the removal of E, D.

Detecting feature model errors: This section aims to propose a method of detecting feature model errors by finding all contradictory relationship sets (CRS) existing in a feature model. For a specific feature f, to identify CRS (f, inclusion), root feature is included and inclusion of f is propagated by feature relationship propagation. Whenever a Contradictory Configuration Conclusion (CCC) arises, a CRS (f, inclusion) is identified. Similarly, to identify CRS (f, removal), the root feature and parent of feature f are included and the removal of f is propagated following feature relationship propagation. Whenever a contradictory configuration conclusion arises, a CRS (f, removal) is identified.

This study implements the feature relationship propagation by Depth-first Search (DFS) in a feature diagram. Starting at a specific feature f, its inclusion or removal is propagated to the first adjacent feature that appears based on the propagation rules. The propagation goes deeper and deeper until a contradictory configuration conclusion is found or until it hits a feature that has no adjacent features that can be propagated. Then the search backtracks returning to the most recent feature it hasn't finished exploring in a non-recursive implementation, all freshly propagated features are added to a stack for exploration.

Algorithm 1 is developed to identify all contradictory relationship sets in a feature model. Algorithm 1 includes two functions: the main function (Function 1) and the function of feature relationship propagation (Function 2). Three global variables are defined: crs_list_f which is an array list for storing the identified CRS starting from f, crs_list which is an array list for storing all CRS existing in a feature model and path which is a stack for tracking the propagation process. In Function 1, to find CRS (f, inclusion), the inclusion of the root feature r and then propagate the inclusion of f. If any contradictory configuration conclusion arises, a CRS (f, inclusion) can be found (line 1-9). To find CRS (f, removal), the inclusion of root feature r and the parent of f is propagated and the removal of f is propagated to other features. If any contradictory configuration conclusion arises, a CRS (f, removal) can be found (line 10-19). The parameter of Function 1 is the feature model (fm) from which finds all CRS. Before identifying CRS starting from a specific feature f, the

feature model must be initialized by the method fm. reset which can set the status of all features in the feature model as "0". The global variable crs_list_f must be cleared to store the CRS starting from f by the method crs_list_f. clear (). All CRS identified will be stored into the array list crs_list as illustrated in line 7 and 17.

Algorithm 1:

```
Global variables:
crs_list_f: the list that stores a set of CRS starting from f
crs_list: the list that stores all CRS in fm
path: the stack that tracks the feature relationship propagation process starting from f
Function 1: Identifying CRS
Parameters:
fm: the given feature model
   Function identifyCRS (fm)
        //identify CRS (f, inclusion)
   1.   Foreach f in fm. variables
   2.       fm. reset ();
   3.       crs_list_f. clear ();
   4.       propagate (r, 1);// r is the root feature of the feature model
   5.       propagate (f, 1);
   6.       Foreach crs_f in crs_list_f
   7.           crs_list. add (new CRS (f, inclusion, crs_f));
   8.       End foreach
   9.   End foreach
        // identify CRS (f, removal)
   10.  Foreach f in fm. variables
   11.      fm. reset ();
   12.      crs_list_f. clear ();
   13.      propagate (r, 1);
   14.      propagate (f. parent, 1);
   15.      propagate (f, -1);
   16.      Foreach crs_f in crs_list_f
   17.          crs_list. add (new CRS (f, removal, crs_f));
   18.      End foreach
   19.      End Foreach
```

In Function 2, the action imposed on a specific feature f is propagated to other features. This function has three parameters: a feature f where the propagation starts from, an action imposed on feature f where value "1" illustrating inclusion of f and value "-1" illustrating the removal of f and the relationship that connects f with its precedent feature which propagates value to f. Before the propagation, feature f is included or removed based on the action imposed on f (line 1) and the feature relationship that connects f with its precedent feature is stored into path to track the propagation process (line 2). In the propagation process, if the action on f can be propagated to an adjacent feature, the function "propagate" is called recursively. If an adjacent feature that has already been included needs to be removed or an adjacent feature that has already been removed needs to be included, a contradictory configuration conclusion is found and the propagation returns to the precedent feature (i.e., line 8, 25). Meanwhile the current propagation path is copied and stored to crs_list_f to present the found contradictory relationship set. If no adjacent features of f

can be removed or included, the propagation returns to the precedent feature and the feature relationship connecting f and its precedent feature is popped from path (line 96). It should be noted that after a CRS is found, the propagation will continue as there may be more than one CRS starting from a feature f. After the propagation, all CRS starting from f will be stored in crs_list_f.

Algorithm 2:

```
Function 2: Feature relationship propagation
Parameters:
f: the feature where the propagation starts from in one propagation step.
action: the configuration action imposed on f, "1" illustrating inclusion of f and "-1" illustrating the removal of f.
relationship: the relationship that connects f with the feature which propagates the value to f.
  Function propagate (f, action, relationship)
  1.   f. setStatus (action);
  2.   path. push (relationship);
       // propagate inclusion of f to its adjacent features
  3.   If (action = = 1)
       // propagate to parent feature
  4.     If (f. parent ! = null andand f. parent. status = = 0)
  5.     propagate (f. parent, 1, getRelation (f, f. parent));
  6.   End If
  7.   If (f. parent ! = null andand f. parent. status = = -1)
  8.     crs_list_f. Add (path. clone ());
  9.   End If
       //propagate to child features
       //getNumofFeatures () takes a variation point as input and returns the number of variant features under the variation point as
       output.
       //getNumofRemoved () takes a variation point as input and returns the number of removed variant features under the variation
       point as output.
  10.    If (f is a variation point)
  11.      If (getNumofRemoved (f) = = getNumofFeatures (f) -f. min_card)
  12.        Foreach child in f. children
  13.          If (child. status = = 0)
  14.            propagate (child, 1, getRelation (f, child));
  15.          End If
  16.      End Foreach
  17.    End If
  18.  Else
  19.      Foreach child in f. children
  20.        If (child is a mandatory feature or variation point)
  21.          If (child. status = = 0)
  22.          propagate (child, 1, getRelation (f, child));
  23.        End If
  24.        If (child. status = = -1)
  25.          crs_list_f. Add (path. clone ());
  26.        End If
  27.      End If
  28.    End Foreach
  29.  End If
       //propagate to brother features
       //getNumofSelected () takes a variation point as input and returns the number of included variant features under the variation
       point as output.
```

Algorithm 2: Continue

```
30.         If (f is variant feature in a variation point andand f. parent. status = = 1)
31.             If (getNumofSelected (f. parent) = = f. parent. max_card)
32.                 Foreach child in f. parent. children
33.                     If (child. status = = 0)
34.                     propagate (child, -1, getRelation (f, child));
35.             End If
36.         End Foreach
37.     End If
38. Else
        //propagate to friend features
39.         Foreach reqs in f. requiresfeatures
40.             If (reqs. status = = 0)
41.                 propagate (reqs, 1, getRelation(f, reqs));
42.             End If
43.             If (reqs. status = = -1)
44.                 crs_list_f. Add (path. clone ());
45.             End If
46.         End Foreach
47.         Foreach excs in f. excludesfeatures
48.             If (excs. status = = 0)
49.                 propagate (excs, -1, getRelation(f, excs));
50.         End If
51.         If (excs. status = = 1)
52.             crs_list_f. Add (path. clone ());
53.         End If
54.     End Foreach
55. End If
            // propagate the removal of f to its adjacent features
56.         If (action = = -1)
            //propagate to parent feature
57.             If (f. parent! = null andand f is a mandatory feature or variation point)
58.                 If (f. parent. status = = 0)
59.                     propagate (f. parent, -1, getRelation (f, f. parent));
60.             End If
61.             If (f. parent. status = = 1)
62.                 crs_list_f. Add (path. clone ());
63.             End If
64.     End If
            // propagate to child feature
65.             Foreach child in f. children
66.                 If (child. status = = 0)
67.                     propagate (child, -1, getRelation(f, child));
68.                 End If
69.                 If (child. status = = 1)
70.                     crs_list_f. Add (path. clone ());
71.                 End If
72.         End Foreach
                // propagate to brother feature
73.                 If (f is variant feature in a variation point andand f. parent. status = = 1)
74.                     If (getNumofRemoved (f. parent) = = getNumofFeatures (f. parent) -f. parent. min_card)
```

Algorithm 2: Continue

```
75.                     Foreach child in f. parent. children
76.                         If (child. status = = 0)
77.                             propagate (child, 1, getRelation (f, child));
78.                         End If
79.                     End Foreach
80.                 End If
81.             End If
82.             If (f is variant feature in a variation point andand f. parent. status = = 0)
83.                 If (getNumofRemoved (f. parent) > getNumofFeatures (f. parent) -f. parent. min_card)
84.                     propagate (f. parent, -1, getRelation (f, f. parent));
85.                 End If
86.             End If
            //propagate to friend feature
87.             Foreach reqed in f. requiredfeatures
88.                 If (reqed. status = = 0)
89.                     propagate (reqed, -1, getRelation(f, reqed));
90.                 End If
81.                 If (reqed. status = = 1)
92.                     crs_list_f. Add (path. clone ());
93.                 End If
94.             End Foreach
95.         End If
96.     path. Pop();
```

Using Algorithm 2, all the contradictory relationship sets (CRS) existing in a feature model can be identified. Based on the relationships between CRS and feature model errors proposed, the feature model errors can be identified based on the found CRS. For example, in the tourist guide feature model of Fig. 1, a CRS {“mobile”, inclusion} can be identified using Algorithm 2, so feature “mobile” is a dead feature. To fix this feature model error, the explanations of the feature model error must be given. The next section will introduce the method of generating the explanations of a feature model error based on the CRS that cause the feature model error.

Explaining feature model errors: Once a feature model error is identified, the explanations of the error need to be provided to correct the error. A feature model error may have a set of explanations and each explanation is a set of feature relationships which must be modified to remove the error. For example, a possible explanation of dead feature “mobile” in Fig. 1 would be the feature dependency “terminal device requires encryption”, which means modifying this feature dependency will remove the dead feature error. Another possible explanation of dead feature “mobile” would be “encryption excludes mobile” and the error can also be corrected by modifying this feature dependency.

The task of explaining a feature model error is a diagnosis problem in the theory of diagnosis (Reiter, 1987). In Reiter’s theory, a diagnosis system is modeled as (SD, COMPS, OBS) where SD is a set of predicates defining the behavioral and structural models of the system, COMPS is the set of system components and OBS is a set of observations expressed as predicates. The abnormal behavior of a component c is represented as Ab (c) while the normal behavior of c is represented as ¬Ab (c). In Reiter’s theory, $\Delta \subseteq$COMPS is an explanation of system (SD, COMPS, OBS) if the predicate “$SD \wedge OBS \wedge \{Ab(c) \mid c \in \Delta\} \wedge \{\neg Ab(c) \mid c \in COMPS-\Delta\}$” is consistent and Δ is minimal. The minimal explanation of a diagnosis system S is defined as an explanation E for S such that no strict

subset of E is also an explanation for S (Poole and Mackworth, 2010). In the context of feature model validation, the minimal explanations of a feature model error are defined as follows:

Definition 1: A minimal explanation of a feature model error is an explanation E of the error such that no strict subset of E is also an explanation of the error, where E is a set of feature relationships that must be modified to remove the error.

In the following, a method is developed to generate the minimal explanations of a feature model error based on the contradictory relationship sets that cause the error. As a feature model error on a specific feature f is caused by one or more contradictory relationship sets, the intuitive idea of fixing a feature model error on f is to eliminate all the contradictory relationship sets that cause the error from the feature model. Assume that a feature model error is caused by a set of contradictory relationship sets CRS_1, CRS_2... CRS_n where "n = 1" means the feature model error is caused by one CRS and "n>1" means the feature model error is caused by multiple CRS. Removing a CRS can be achieved by modifying one or more feature relationships in the CRS. Therefore, there can be several solutions of removing a CRS by changing at least one feature relationship in the CRS. Among these solutions, the optimal solutions of removing a CRS should include one and only one feature relationship in the CRS. Deductively, the optimal solutions of removing CRS_1, CRS_2... CRS_n should include one feature relationship from each CRS and they can be obtained from the cartesian product of CRS_1, CRS_2... CRS_n ($CRS_1 \times CRS_2 \times ... \times CRS_n = \{(r_1, r_2..., r_n): r_i \in CRS_i\}$). To generate the minimal explanations of a feature model error caused by CRS_1, CRS_2... CRS_n, four steps need to be followed:

Step 1: Obtain all solutions of removing CRS_1, CRS_2... CRS_n by $MS = CRS_1 \times CRS_2 \times ... \times CRS_n = \{s_1, s_2... s_m\}$ where $m = Card(CRS_1) \times Card(CRS_2) \times ... \times Card(CRS_n)$

Step 2: Combine the repeated elements in a solution s_i ($s_i \in MS$), as an element can only appear once in a solution.

Step 3: Remove the solution s_x from MS if another solution s_y is the strict subset of s_x (s_x, $s_y \in MS$) based on Definition 1

Step 4: Use the remaining solutions in MS as the minimal explanations of the feature model error caused by CRS_1, CRS_2... CRS_n

The dead feature *A* in feature model of Fig. 5 is taken as an example to illustrate the process of generating the minimal explanations of a feature model error from its related contradictory

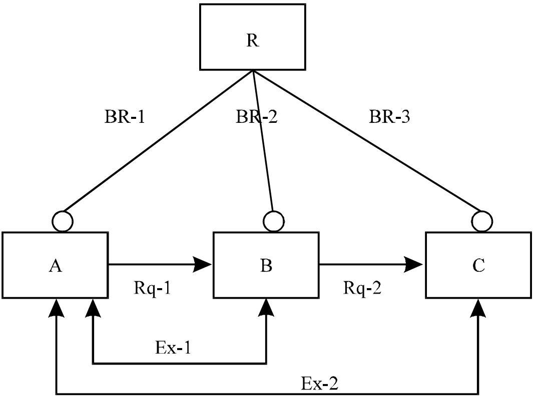

Fig. 5: An example for illustrating minimal explanations

relationship sets. First, Algorithm 1 is used to find the contradictory relationship sets that cause dead feature A: CRS_1 = (Rq-1, Ex-1) and CRS_2 = (Rq-1, Rq-2, Ex-2). Then all optimal solutions of removing CRS_1 and CRS_2 are calculated, s_1 = (Rq-1, Rq-1), s_2 = (Rq-1, Rq-2), s_3 = (Rq-1, Ex-2), s_4 = (Ex-1, Rq-1), s_5 = (Ex-1, Rq-2), s_6 = (Ex-1, Ex-2) by $CRS_1 \times CRS_2$ As s_1 has repeated elements, it is combined into (Rq-1). The solutions s_2, s_3 and s_4 are removed from the solution list as s_1 = (Rq-1) is their strict subset. Finally the remaining solutions s_1 = (Rq-1), s_5 = (Ex-1, Rq-2) and s_6 = (Ex-1, Ex-2) can be used as the minimal explanations of dead feature A.

In Reiter's theory, the error explanation for dead feature A in Fig. 5 can be represented as a diagnosis system (SD', COMPS', OBS') where the predicate "SD'∧OBS'∧{Ab(c') | c'∈**Δ**}∧{¬Ab(c') | c'∈COMPS'-**Δ**}". Here, SD' includes the rules of constraining feature selections in the feature model and these rules arise from all feature relationships; COMPS' includes all feature relationships in the feature model and OBS' indicates the inclusions or removals of some specific features. Ab (c') means that feature relationship c' should be absent in the feature model and ¬ Ab (c') means that the feature relationship c' should be present in the feature model. To find the explanations of dead feature A, the predicate in OBS' would be the inclusion of feature A. As the found minimal explanations of dead feature A: **Δ** = (Rq-1), **Δ** = (Ex-1, Rq-2) and **Δ** = (Ex-1, Ex-2) can make the predicate "SD'∧OBS'∧{Ab(c') | c'∈**Δ**}∧{¬Ab(c') | c'∈COMPS'-**Δ**}" consistent, the minimal explanations identified by the proposed approach are the right explanations of the feature model error in Reiter's theory. Table 4 shows some examples to illustrate the process of generating the minimum explanations for a feature model error based on the contradictory relationship sets that cause the feature model error.

TOOL SUITE

To enhance the usability of the validation approach proposed in this study, a tool suite called Feature Model Validation Tool (FMVTool) has been developed. This tool suite includes two main

Table 4: Examples of generating minimal explanations based on contradictory relationship set

Feature model	Feature model error and CSR	Generating minimal explanations Step 1	Step2	Step 3 and 4 (minimal explanation)
	Dead feature B	{Rq-1, Rq-1, Ex-3}	{Rq-1, Ex-3}	{Rq-1, Ex-3}
		{Rq-1, Rq-1, Rq-3}	{Rq-1, Rq-3}	{Rq-1, Rq-3}
	CRS (B, inclusion) =	{Rq-1, Ex-1, Ex-3}	{Rq-1, Ex-1, Ex-3}	{Ex-2, Ex-1, Ex-3}
	{Rq-1, Ex-2}	{Rq-1, Ex-1, Rq-3}	{Rq-1, Ex-1, Rq-3}	{Ex-2, Ex-1, Rq-3}
		{Rq-1, Rq-2, Ex-3}	{Rq-1, Rq-2, Ex-3}	{Ex-2, Rq-2, Ex-3}
	CRS (B, inclusion) =	{Rq-1, Rq-2, Rq-3}	{Rq-1, Rq-2, Rq-3}	{Ex-2, Rq-2, Rq-3}
	{Rq-1, Ex-1, Rq-2}	{Ex-2, Rq-1, Ex-3}	{Ex-2, Rq-1, Ex-3}	
		{Ex-2, Rq-1, Rq-3}	{Ex-2, Rq-1, Rq-3}	
		{Ex-2, Ex-1, Ex-3}	{Ex-2, Ex-1, Ex-3}	
	CRS (B, inclusion) =	{Ex-2, Ex-1, Rq-3}	{Ex-2, Ex-1, Rq-3}	
	{Ex-3, Rq-3}	{Ex-2, Rq-2, Ex-3}	{Ex-2, Rq-2, Ex-3}	
		{Ex-2, Rq-2, Rq-3}	{Ex-2, Rq-2, Rq-3}	
	Dead feature B			
	CRS (B, inclusion) =	{Rq-1, Rq-2}	{Rq-1, Rq-2}	{Rq-1, Rq-2}
	{Rq-1, Ex-1}	{Rq-1, Ex-2}	{Rq-1, Ex-2}	{Rq-1, Ex-2}
	CRS (B, inclusion) =	{Ex-1, Ex-2}	{Ex-1, Ex-2}	{Ex-1, Ex-2}
	{Rq-2, Ex-2}	{Rq-2, Ex-1}	{Rq-2, Ex-1}	{Rq-2, Ex-1}

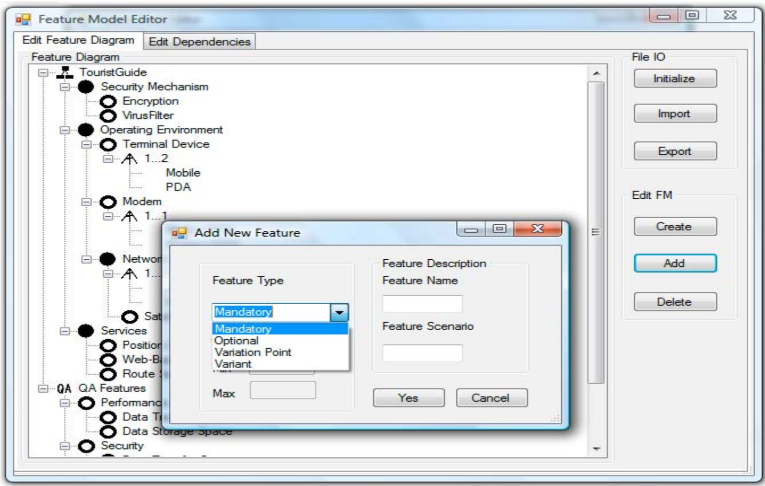

Fig. 6: View of constructing feature models in feature model validation tool

```
<?xml version = "1.0" encoding = "GB2312" ?>
- <feather-model>
- <feather-name = " ">
- <binaryRelation name = " ">
<cardinality min = " " max = " "/>
<solitaryFeather name = " "/>
</binaryRelation>
- <setRelation name = " ">
<cardinality min = " " max = " "/>
<groupedFeather name = " "/>
<groupedFeather name = " "/>
</setRelation>
</feather>
<qaFeather name = " "/>
<requires name = " " feather = " " requires = " "/>
<excludes name = " " feather = " " excludes = " "/>
</feather-model>
```

Fig. 7: XML Format for storing feature models in feature model validation tool

functions: generating feature models and validating feature models. The FMVTool provides two ways for generating feature models. First, the model designers can use FMVTool to build a feature model manually. Figure 6 shows the view of establishing a feature model by the designers. The FMVTool supports to create a root feature and add mandatory feature, optional feature, variation point feature or variant feature under an existing feature. The “requires” or “excludes” dependencies can be added between two variable features. Second, FMVTool can generate the random feature models based on a set of parameters, such as the number of features, the number of cross-tree constraints, the percentage of common features and variable features among all features and the maximum height of feature models. The FMVTool provides the function of generating random feature models, as the efficiency of the proposed validation approach needs to be evaluated based on large scale feature models which are difficult to obtain in real world software product lines.

Once a feature model is established, it needs to be stored into files. As shown in Fig. 7, the xml format that FAMA (Benavides *et al*., 2007) used is adapted for storing feature models. The

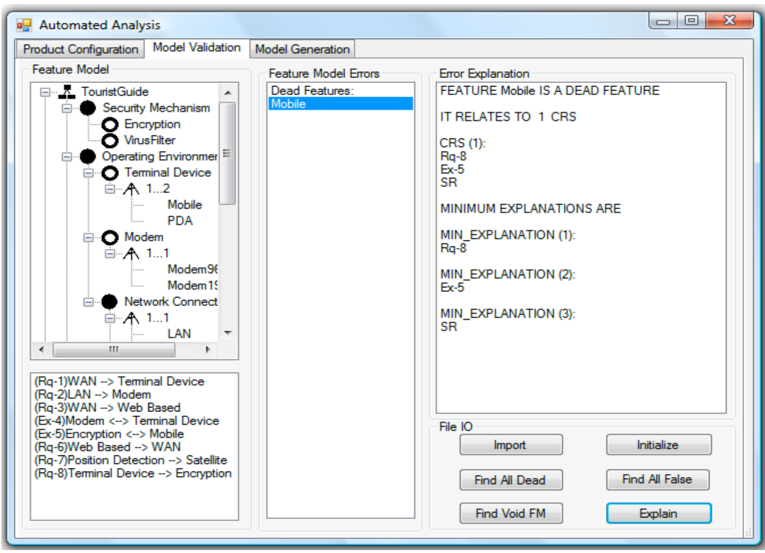

Fig. 8: View of debugging feature model in feature model validation tool

<binaryRelation> with <cardinality min = "0", max = "1"> represents the optional relationships and the <binaryRelation>with<cardinality min = "1", max = "1"> represents mandatory relationships. The <solitaryFeature>presents a child feature that connects with its parent feature by optional or mandatory relationship. The <setRelation> with <cardinality min = "a", max = "b"> represents a variation point with cardinality (a... b). Each variant feature under a variation point is represented by <groupedFeature> under <setRelation>. The "requires" and "excludes" constraints are described by the elements <requires> and <excludes> respectively.

To validate a feature model, FMVTool supports to identify feature model errors and explain a specific feature model error. Figure 8 shows the view of identifying and explaining feature model errors in FMVTool. An existing feature model is first input into FMVTool from an xml file. Then dead features can be identified by pressing "find all dead" and false variable features can be identified by pressing "find all false". The found dead features and false variable features will appear in the list box of "feature model errors". To explain a specific feature model error on f, feature f is selected in the list box of "feature model errors" and press "explain". Then all Contradictory Relationship Sets (CRS) that cause the feature model error and the minimal explanations of the feature model error will be shown in the list box of "error explanation". Figure 8 shows the results of using FMVTool to validate the tourist guide feature model of Fig. 1. The result shows that "mobile" is a dead feature because of CRS ("mobile", inclusion) = {Rq-8, Ex-5, SR}. This feature model error has three minimal explanations {Rq-8}, {Ex-5} and {SR}. Here, the explanation "SR" is the hierarchical relationship between "terminal device" and "mobile". The names of hierarchical relationships can be found in the xml file which stores the feature model.

Finally, feature modelers can fix the feature model errors by modifying feature relationships based on the error explanations found by FMVTool. These feature relationships can be modified in two ways: modifying the xml file which stores the feature model and modifying the feature model from the function view of Fig. 6.

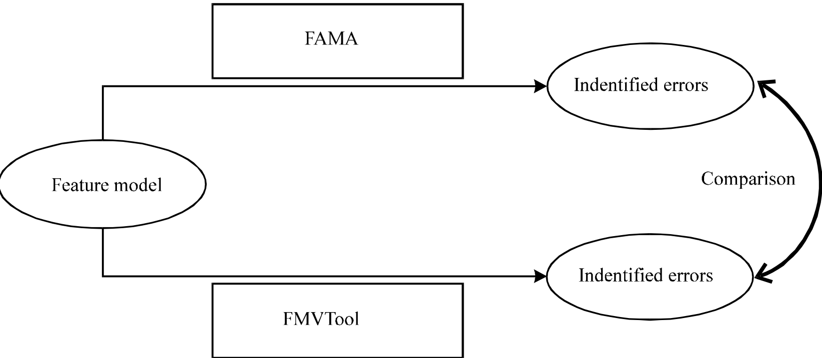

Fig. 9: Evaluation process for correctness and efficiency of proposed approach

EXPERIMENTS

This section aims to evaluate the proposed validation approach from two aspects: the correctness of the identified feature model errors and the efficiency for validating feature models. The evaluation process is illustrated in Fig. 9. The CSP based approach proposed by Trinidad *et al.* (2008) and its supporting tool FAMA (Benavides *et al.*, 2007) are chosen as the contrast to evaluate the approach of this study and the tool suite FMVTool. For a specific feature model, FAMA and FMVTool are used to identify feature model errors and find the explanations for each feature model error. Then the validation results from FAMA and FMVTool can be compared.

CORRECTNESS

The correctness of the proposed approach heavily relies on whether the identified feature model errors and their explanations are complete and correct. To evaluate the correctness of the proposed approach, FMVTool and FAMA are used to validate a set of feature models as shown in Table 5 and then compare the validation results. All kinds of feature model errors are included into these examples, such as dead feature and false variable feature. Furthermore, these examples include feature model errors caused by one Contradictory Relationship Set (CRS), by multiple CRS that have no interactions and by multiple CRS that have intersections. The validation results show that FMVTool can identify feature model errors and provide the explanations for feature model errors the same as FAMA. It should be noted that the explanations identified by FMVTool can cover the explanations identified by FAMA. FMVTool finds the minimal explanations based on Definition 1. FAMA finds the minimum explanation which is an explanation that contains the minimum set of failing feature relationships while giving the reason of the error (Trinidad *et al.*, 2008). For example, in the third feature model of Table 5, FAMA can identify the minimum explanation (Rq-1) for dead feature A while FMVTool can identify the minimal explanations (Rq-1) (Ex-1, Rq-2) and (Ex-1, Ex-2) for dead feature A.

The next step is to validate the developed tool FMVTool on some randomly generated feature models. The numbers are used to name the features in the generated feature models. For example, if a feature model with 5 features is generated, the names of the five features in the feature model are 1, 2, 3, 4 and 5. In the experiment, 15 random feature models are generated as shown in Table 6. In these feature models, the number of features ranges from 20 to 40 and the number of feature dependencies ranges from 10 to 20.

Table 5: Comparison of validation result from feature model analyzer (FAMA) and feature model validation tool (FMVTool)

Feature models	Validation result in FAMA	Validation result in FMVTool
	Dead Feature: C3	Vaildation results in Fmv Tool
	Minimum explanations:	Feature C3 is a dead feature
	SR-1	It relates to 1 CRS
	Ex-1	CRS (1):
	Rq-1	Ex-1
		SR-1
		Min_ Explantation (1):
		Rq-1
		Min_Explanation (2):
		Ex-1
		Min_ Explanatio (3):
		SR-1
	False-mandatory Feature: D	Featured is aflase variable feature
	Minimum explanations:	It relates to 1 CRS
	SR-1	CRS (1):
	Rq-1	BR-2
	BR-2	Rq-1
	Ex-1	Ex-1
		SR-1
		Min_ Explanation (1):
		Br-2
		Min_Explanation (2):
		Rq-1
		Min_Explantion (3):
		Ex-1
		Min_Explanation (4)
		SR-1
	Dead Feature: A	Feature a is a dead feature
	Minimum explanations:	It relates to 2 CRS
	Rq-1	CRS (1):
		Rq-2
		Ex-2
		Rq-1
		CRS (2):
		Rq-1
		Ex-1
		Min_Explanation (2):
		Ex-1
		Ex-2
		Min_Explanation (3):
		Ex-1
		Rq-2
	Dead Feature: B	Feature B is adead featur
	Minimum explanations:	It relates to CRS
	Ex-2 Ex-1	CRS (1):
	Ex-2 Rq-1	Rq-1
	Ex-1 Rq-2	Ex-1
	Rq-1 Rq-2	CRS (2):

Table 5: Continue

Feature models	Validation result in FAMA	Validation result in FMVTool
		Ex-2
		Rq-2
		Min_Explanation (1):
		Ex-2
		Rq-1
		Rq-2
		Min_Explanation (3):
		Ex-2
		Ex-1
		Min_Explanation (4):
		Rq-2
		Ex-1

Table 6: Identified feature model errors in randomly generated feature models

Feature model	No. of features	No. of requires	No. of excludes	Dead feature error	False variable feature
1	20	5	5	5, 18, 11, 12, 17	
2	20	5	5	13	2
3	20	5	5	20	
4	30	10	10	18, 19, 27	
5	30	10	10	20, 25, 30	10, 19
6	30	10	10	4, 15, 16, 17, 22, 25	
7	30	10	10	9, 25, 26, 27, 28, 29, 30, 15, 2, 7, 23, 24, 8	
8	30	10	10		
9	30	10	10	21, 7, 24, 4, 16, 17	
10	30	10	10	5, 24, 25, 28, 16	23
11	40	10	10	28	
12	40	10	10	37, 18, 20, 21, 22, 24, 25, 27	26
13	40	10	10	30, 9, 34, 12, 35, 36, 37, 38	
14	40	10	10	11, 36, 13, 22, 28	3, 18
15	40	10	10	6, 25, 26, 29, 32, 33, 35, 36, 13	

Table 6 shows the results of using FMVTool to identify dead features and false variable features in the generated feature models. By comparing the validation results of the generated feature models provided by FAMA with the validation results in Table 6, FMVTool and FAMA can identify the same set of feature model errors for these randomly generated feature models. Furthermore, it is found that the minimal explanations generated by FMVTool can equal with or cover the minimum explanations provided by FAMA on these generated feature models. For example, Fig. 10 shows the random feature model 2 in Table 6. FMVTool can identify the minimal explanations of dead feature 13 as (Rq-2) (BR-7) (BR-8) (Rq-1) and (Ex-1); and the minimal explanations of false variable feature 2 as (Rq-3) (SR-4) (Rq-4) and (Rq-5). This result is the same as the validation result from FAMA. In this sense, FMVTool can identify and explain feature model errors correctly on these randomly generated feature models.

Efficiency: This section aims to verify the efficiency of the proposed validation approach based. 5 groups of feature models are randomly generated and each group consists of 10 feature models.

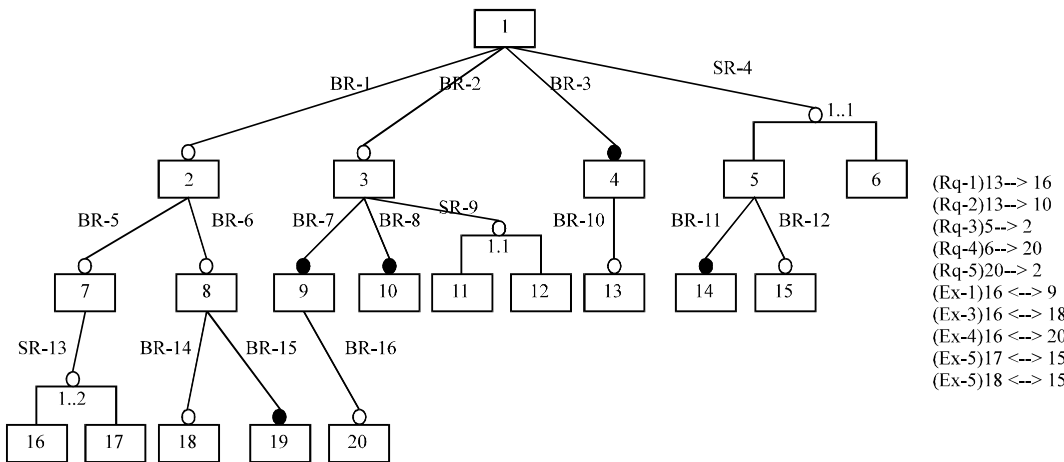

Fig. 10: A random feature model in Table 6

Table 7: The time spent for identifying feature model errors by feature model analyzer (FAMA) and feature model validation tool (FMVTool)

Feature model group	Features	Requires	Excludes	FMVTool (msec)	FAMA (msec)
1	20	5	5	100	6968
2	30	10	10	160	10200
3	40	10	10	320	15600
4	100	10	10	610	Error
5	1000	50	50	1180	Error

The feature models in the same group have the same number of features and the same number of cross-tree constraints. Then FMVTool and FAMA are used to validate 5 groups of randomly generated feature models. The average time spent by FAMA and the average time spent by FMVTool for validating each group of feature models are recorded in Table 7. The results show that FAMA reveals a weak time performance when resolving feature model errors on medium and large size feature models. For validating a feature model which has more than 100 features, FAMA even meets “out of memory” error because of the state space explosion problem. Compared with FAMA, the proposed validation approach reveals a strong time performance and finds the explanations for feature model errors in seconds.

Besides the experiments, the efficiency of the proposed approach is further compared with the CSP based approach from the algorithm complexity. As introduced in section 4, the complexity of the proposed approach is mainly determined by the propagation algorithm (Algorithm 1) which can find all contradictory feature relationships in a feature model. The process of finding contradictory feature relationships from a feature model is the same as the process of finding all elementary circuits from a directed graph (Tarjan, 1972). The time complexity of the developed algorithm can have an upper bound of O(v+e) (c+1) on time and an upper bound of O(v+e) on space when applied to a feature model with v features, e feature relationships and c elementary circuits (Johnson, 1975). If the number of the Contradictory Relationship Sets (CRS) is not large, the approach of this study is more efficient than the CSP based approach which has the time complexity O (2^n).

In conclusion, we observe that our validation tool can produce the same results as FAMA for the same test cases (a set of pre-designed feature models and randomly generated feature models). This observation increases the level of confidence on the correctness of our proposed validation approach. For the validation performance, we find that our validation tool never fails on the test cases and needs a few seconds to generate the validation results for large-scale feature models (more than 100 features).

CONCLUSION

A feature model is one of the most important domain models for capturing and managing the commonalities and variabilities of products in a software product line. In software product line engineering, a feature model is established by domain engineers based on their knowledge and experience. In practice, it is inevitable for domain engineers to introduce the errors into a feature model, especially for large-scale feature models. In order to produce the error-free feature models, a feature model needs to be validated, such as identifying feature model errors and providing the explanations for each identified error. In literature, a set of approaches have been proposed to validate a feature model by transforming a feature model into a constraint satisfaction problem (CSP) and then using solvers to reason on the CSP. However, all these approaches are inefficient when dealing with the large scale feature models, as the time complexity of using the solvers to identify the explanations of a given feature model error is $O(2^n)$ in the general cases. This study proposed a new approach for identifying and explaining feature model errors by finding the contradictory feature relationships behind the feature model errors. A tool suite was developed to implement the concepts of the proposed approach. The correctness and efficiency of the proposed approach have been evaluated by comparing the validation results between FAMA and the developed tool FMVTool. The evaluation showed that the proposed approach is more efficient than the CSP based approach in most cases where the number of contradictory relationship sets is not large. This study focused on validating cardinality-based feature models. In the future, the proposed approach will be extended to validate feature models based on other notations, such as feature models with attributes.

REFERENCES

Asikainen, T., T. Mannisto and T. Soininen, 2006. A unified conceptual foundation for feature modelling. Proceedings of the 10th International Software Product Line Conference, August 21-24, 2006, Baltimore, Maryland, USA., pp: 31-40.

Batory, D., D. Benavides and A. Ruiz-Cortes, 2006. Automated analysis of feature models: Challenges ahead. Commun. ACM, 49: 45-47.

Benavides, D., P. Trinidad and A.Ruiz-Corths, 2005. Automated reasoning on feature models. Proceedings of the 17th Conference on Advanced Information System Engineering, June 13-17, 2005, Porto, Portugal, pp: 491-503.

Benavides, D., S. Segura and A. Ruiz-Cortes, 2010. Automated analysis of feature models 20 years later: A literature review. Inform. Syst., 35: 615-636.

Benavides, D., S. Segura, P. Trinidad and A. Ruiz-Cortes, 2007. FAMA: Tooling a framework for the automated analysis of feature models. Proceedings of the 1st International Workshop on Variability Modelling of Software Intensive Systems, January 16-18, 2007, Limerick, Ireland.

Clements, P. and L. Northrop, 2002. Software Product Lines: Practices and Patterns. Addison-Wesley, USA.

Cook, S.A., 1971. The complexity of theorem-proving procedures. Proceedings of the 3rd annual ACM symposium on Theory of computing, May 3-5, 1971, New York, USA., pp: 151-158.

Czarnecki, K. and P. Kim, 2005. Cardinality-based feature modeling and constraints: A progress report. Proceedings of the International Workshop on Software Factories, October 16-20, 2005, San Diego, California, USA.

Czarnecki, K., S. Helsen and U. Eisenecker, 2004. Staged configuration using feature models. Proceedings of the 3rd International Conference on Software Product Lines, August 30-September 2, 2004, Boston, MA., USA., pp: 266-283.

Johnson, D.B., 1975. Find all elementary circuits of a directed graph. SIAM J. Comput., 4: 77-84.

Kang, K.C., S.G. Cohen, J.A. Hess, W.E. Novak and A.S. Peterson, 1990. Feature-Oriented domain analysis (FODA) feasibility study. Technical Report, CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute.

Mackworth, A.K., 1977. Consistency in networks of relations. Artif. Intell., 8: 99-118.

Mendonca, M., 2009. Efficient reasoning techniques for large scale feature models. Ph.D. Thesis, University of Waterloo, Waterloo, Ontario, Canada.

Mendonca, M., D. Cowan, W. Malyk and T. Oliveira, 2008. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. J. Software, 3: 69-82.

Poole, D.L. and A.K. Mackworth, 2010. Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press, Cambridge, UK., ISBN: 9780521519007, Pages: 682.

Rabiser, R., 2009. A user-centered approach to product configuration in software product line engineering. Ph.D. Thesis, Christian Dopper Laboratory for Automated Software Engineering. Linz, Austria, Institute for Systems Engineering and Automation.

Rabiser, R., P. O'Leary and I. Richardson, 2011. Key activities for product derivation in software product lines. J. Syst. Softw., 84: 285-300.

Reiter, R., 1987. A theory of diagnosis from first principles. Artif. Intell., 32: 57-95.

Riebisch, M., K. Bollert, D. Streitferdt and I. Philippow, 2002. Extending feature diagrams with UML multiplicities. Proceedings of the 6th Conference on Integrated Design and Process Technology, June 23-28, 2002, Pasadena, California, USA., pp: 1-7.

Segura, S., 2008. Automated analysis of feature models using atomic sets. Proceedings of the 1st Workshop on Analyses of Software Product Lines, September 12, 2008, Limerick, Ireland, pp: 201-207.

Tarjan, R.E., 1972. Enumeration of the elementary circuits of a directed graph. SIAM J. Comput., 2: 211-216.

Trinidad, P., D. Benavides, A. Duran, A. Ruiz-Cortes and M. Toro, 2008. Automated error analysis for the agilization of feature modeling. J. Syst. Software, 81: 883-896.

Von der MaBen, T. and H. Lichter, 2004. Deficiencies in feature models. Proceedings of the Workshop on Software Variability Management for Product Derivation, August, 2004, Boston, MA., USA.

Yan, H., W. Zhang, H. Zhao and H. Mei, 2009. An optimization strategy to feature models verification by eliminating verification-irrelevant features and constraints. Proceedings of the 11th International Conference on Software Reuse, September 27-30, 2009, Falls Church, VA., USA., pp: 65-75.

Ye, H., 2005. Approach to modelling feature variability and dependencies in software product lines. IEE Proc. Software, 152: 101-109.

Ye, H., Y. Lin and W. Zhang, 2010. Streamlined feature dependency representation in software product lines. Proceedings of the International Conference on Software Engineering Research and Practice, July 12-15, 2010, Las Vegas, Nevada, pp: 612-618.

Zhang, W., H. Zhao and H. Mei, 2004. A propositional logic-based method for verification of feature models. Proceedings of the 6th International Conference on Formal Engineering Methods, November 8-12, 2004, Seattle, WA., USA., pp: 115-130.